P&S Heterogeneous Systems

Parallel Patterns: Sparse Matrices

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich

Fall 2022

12 December 2022

Parallel Patterns

Reduction Operation

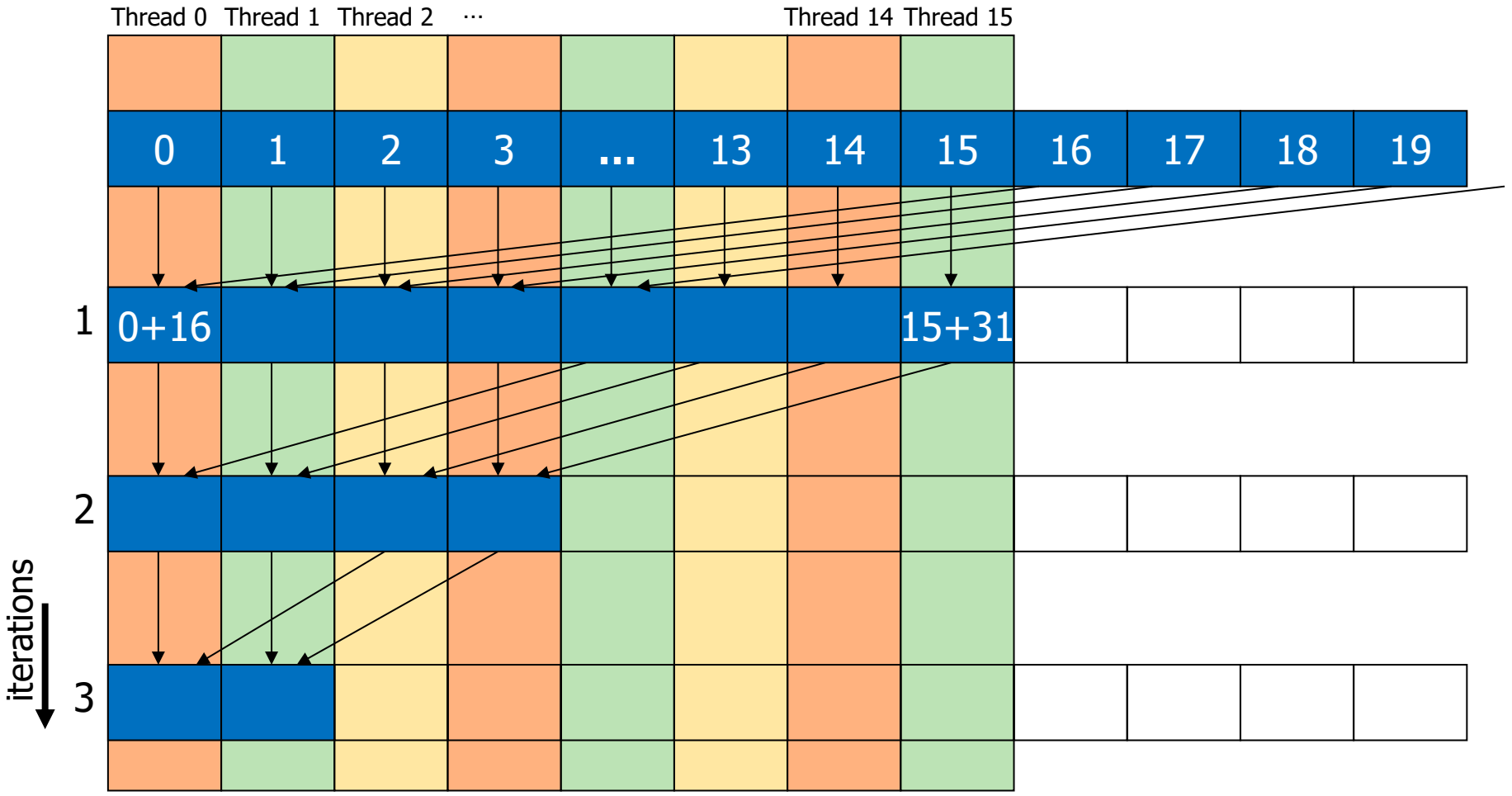
- A reduction operation reduces a set of values to a single value
  - Sum, Product, Minimum, Maximum are examples

- Properties of reduction
  - Associativity
  - Commutativity
  - Identity value

- Reduction is a key primitive for parallel computing
  - E.g., MapReduce programming model

Divergence-Free Mapping (I)

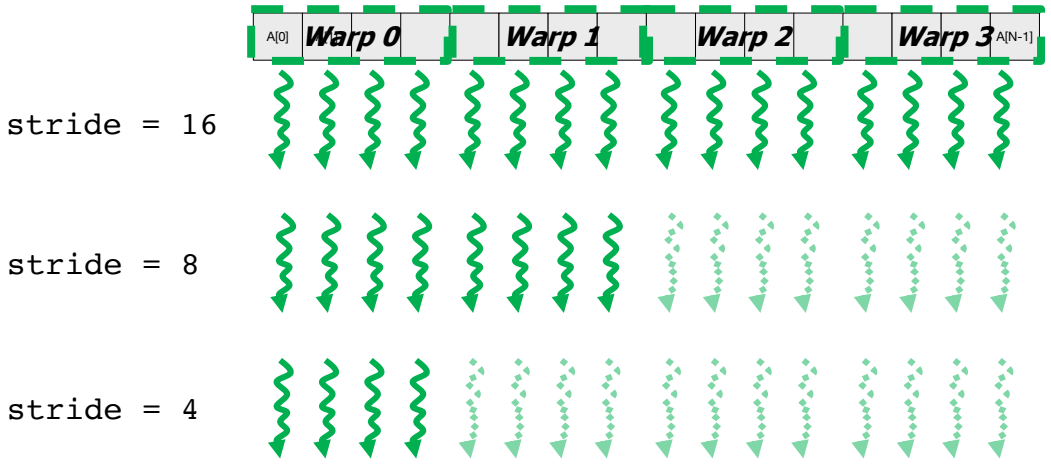
- All active threads belong to the same warp



Divergence-Free Mapping (II)

- Program with high SIMD utilization

```
__shared__ float partialSum[]

unsigned int t = threadIdx.x;

for(int stride = blockDim.x; stride > 0;  stride >> 1){

  __syncthreads();

  if (t < stride)
    partialSum[t] += partialSum[t + stride];

}
```



Warp utilization is maximized

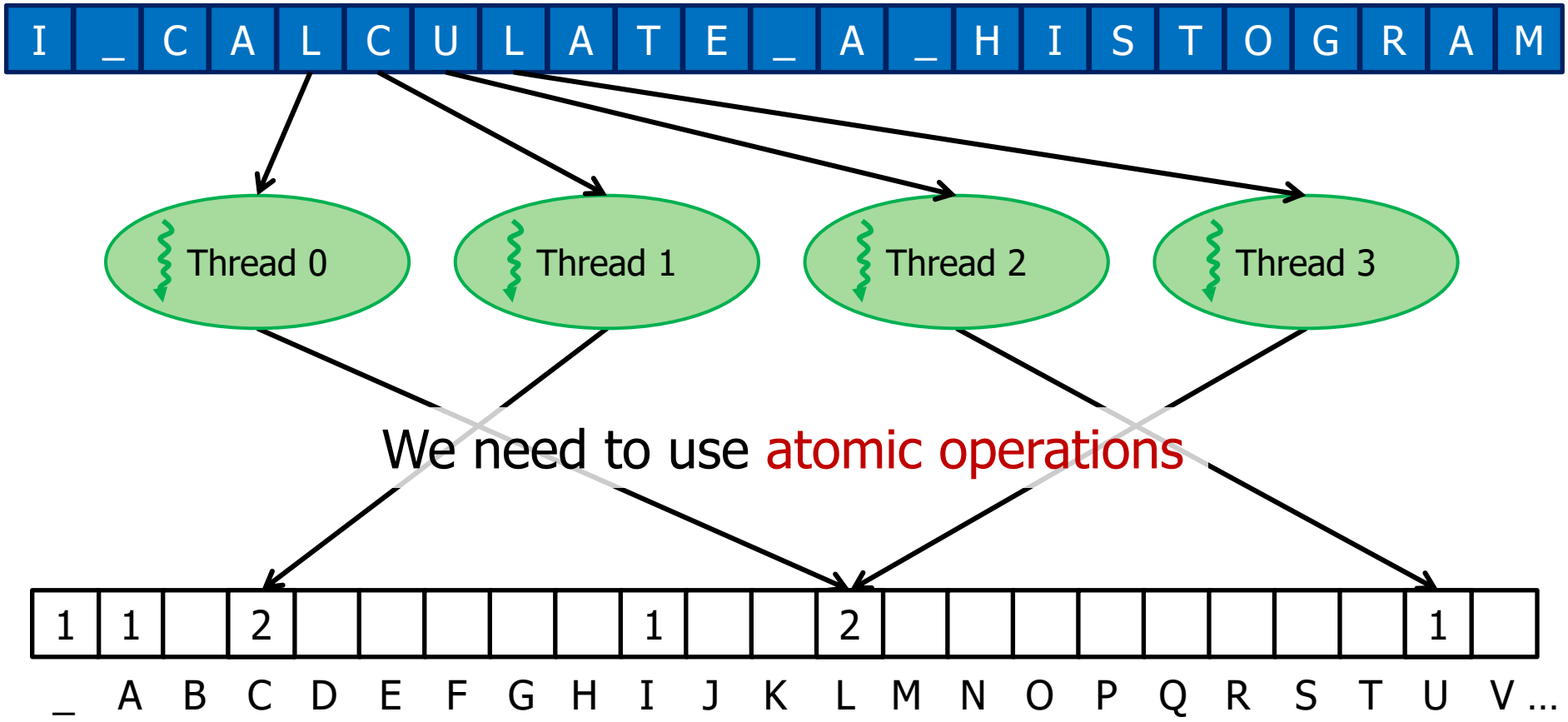
Histogram Computation

- Histogram is a frequently used computation for reducing the dimensionality and extracting notable features and patterns from large data sets
  - Feature extraction for object recognition in images
  - Fraud detection in credit card transactions
  - Correlating heavenly object movements in astrophysics
  - …
- Basic histograms - for each element in the data set, use the value to identify a "bin" to increment
  - Divide possible input value range into "bins"
  - Associate a counter to each bin
  - For each input element, examine its value and determine the bin it falls into and increment the counter for that bin

Slide credit: Hwu & Kirk

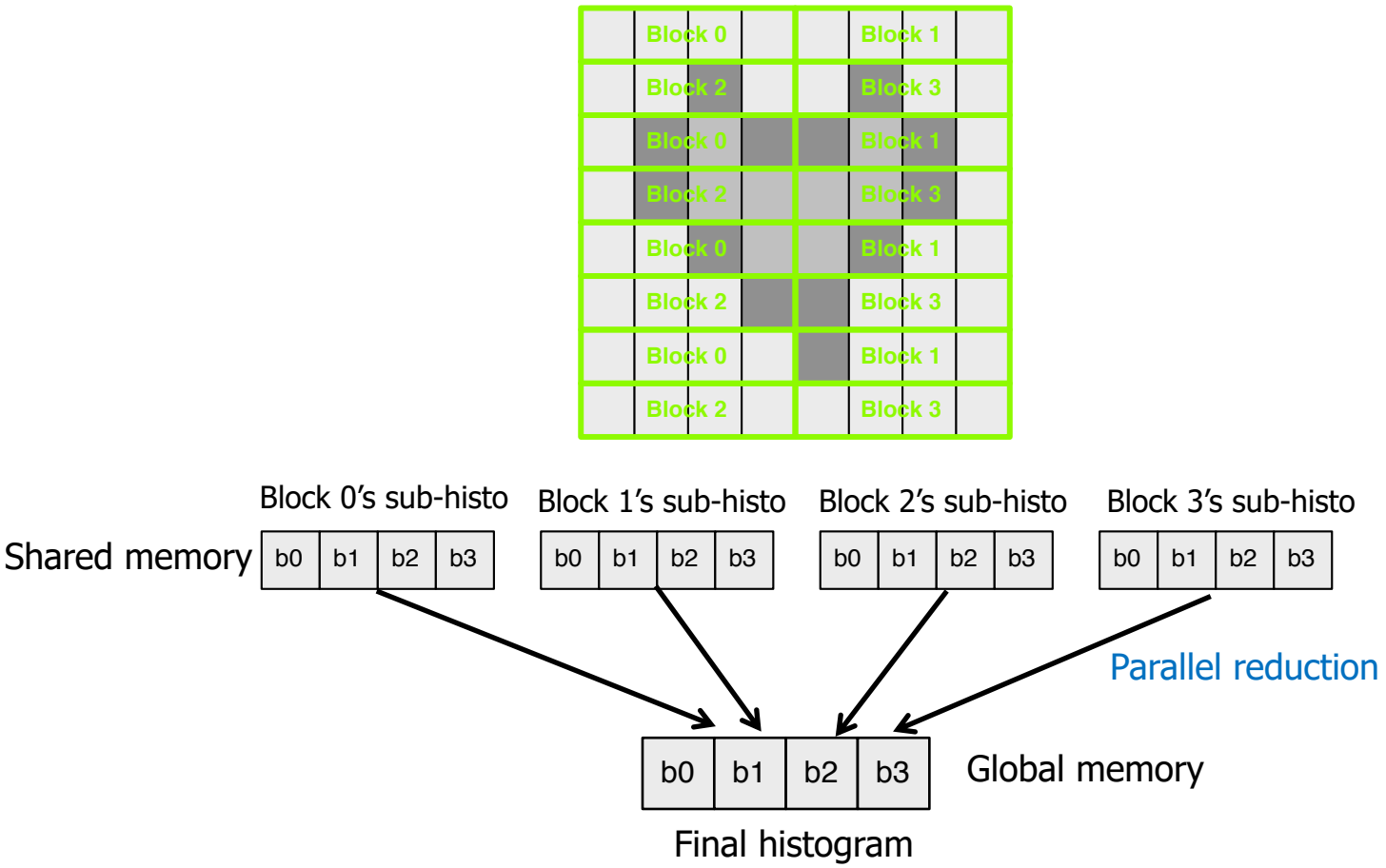
Parallel Histogram Computation: Iteration 2

- All threads move to the next section of the input
  - Each thread moves to element threadID + #threads



Histogram Privatization

- Privatization: Per-block sub-histograms in shared memory
  - Threads use atomic operations in shared memory



Convolution Applications

- Convolution is a widely-used operation in signal processing, image processing, video processing, and computer vision

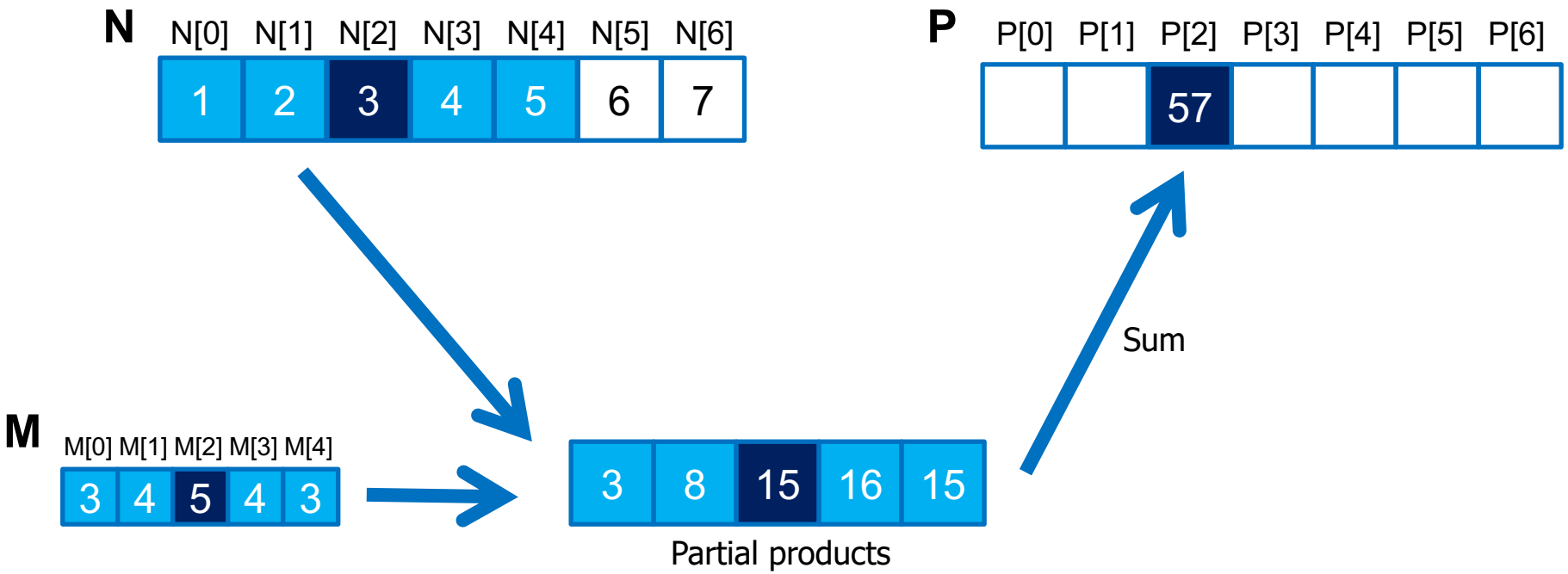
- Convolution applies a filter or mask or kernel* on each element of the input (e.g., a signal, an image, a frame) to obtain a new value, which is a weighted sum of a set of neighboring input elements
  - Smoothing, sharpening, or blurring an image
  - Finding edges in an image
  - Removing noise, etc.

- Applications in machine learning and artificial intelligence
  - Convolutional Neural Networks (CNN or ConvNets)

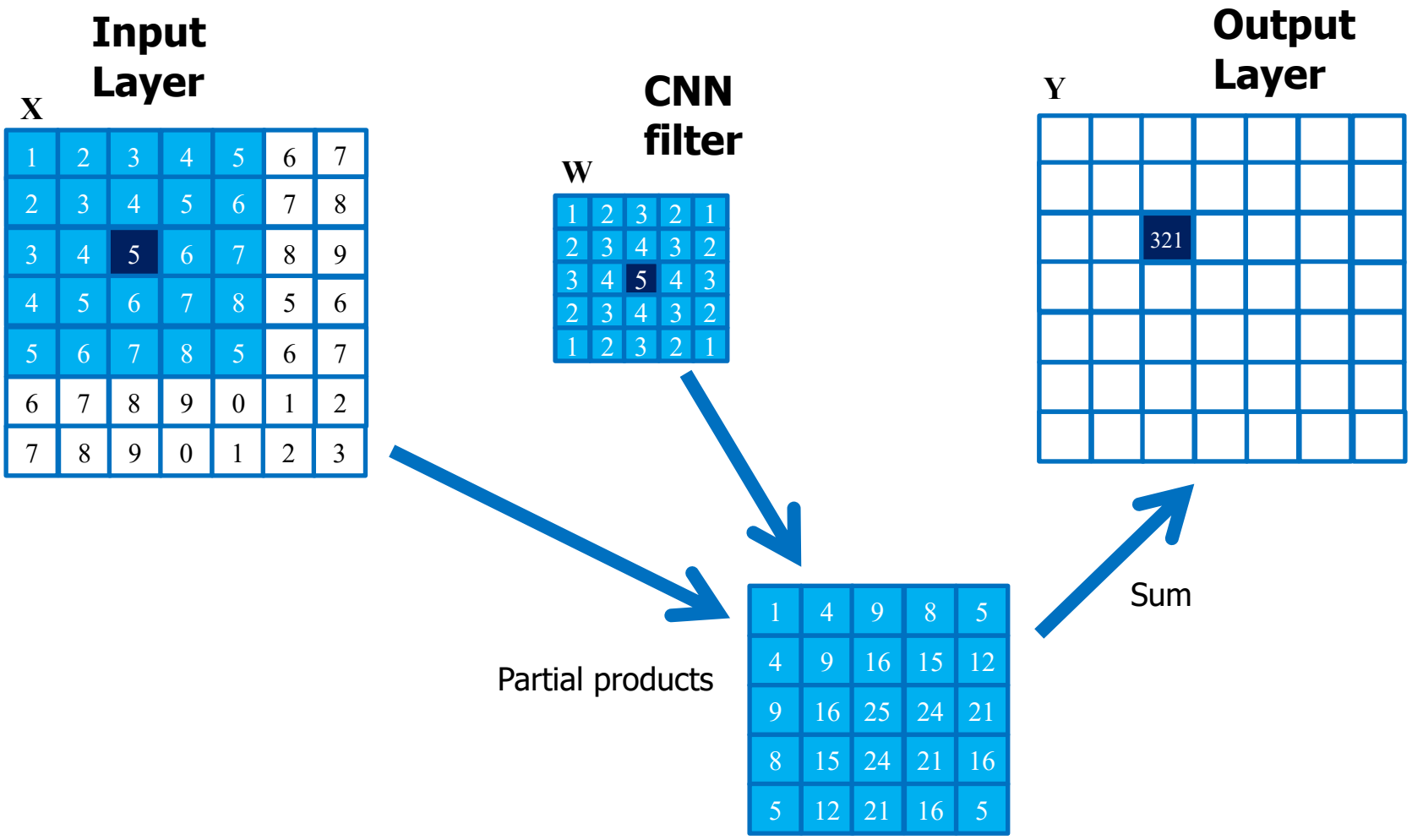
* The term "kernel" may create confusion in the context of GPUs (recall a CUDA/GPU kernel is a function executed by a GPU)

1D Convolution Example

- Commonly used for audio processing
- Mask size is usually an odd number of elements for symmetry (5 in this example)
- Calculation of P[2]:

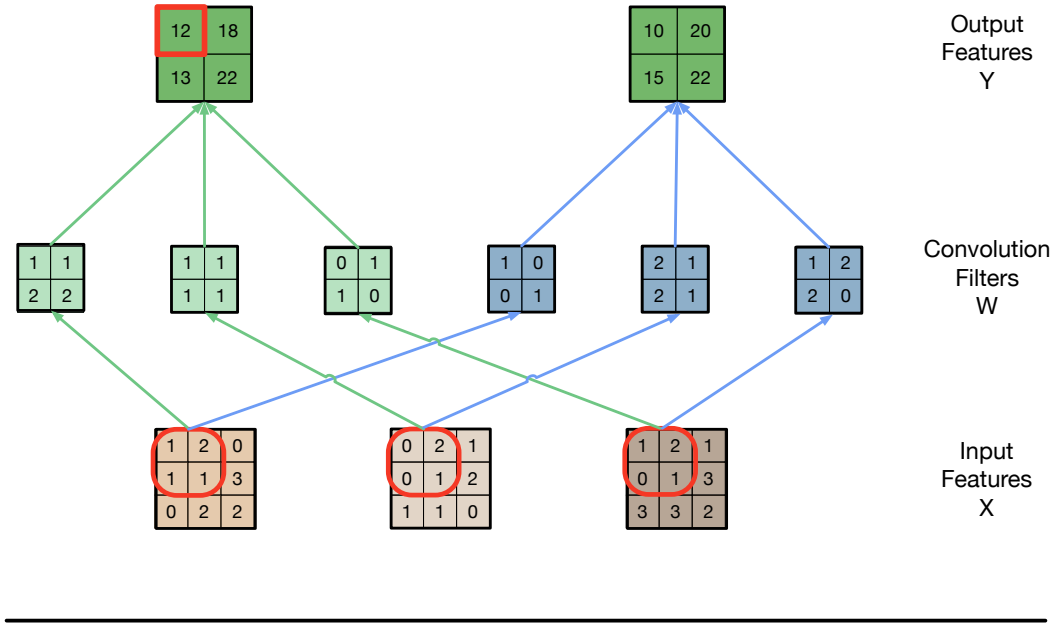


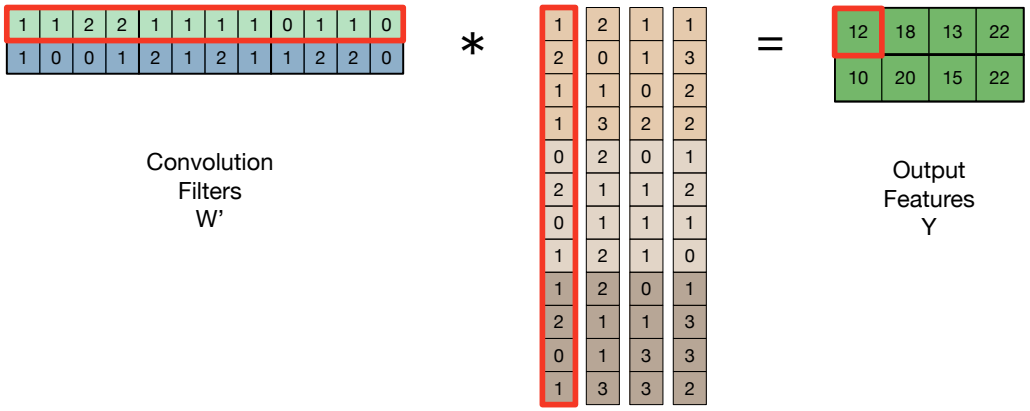
Another Example of 2D Convolution



Slide credit: Hwu & Kirk

Implementing a Convolutional Layer with Matrix Multiplication





Slide credit: Reproduced from Hwu & Kirk

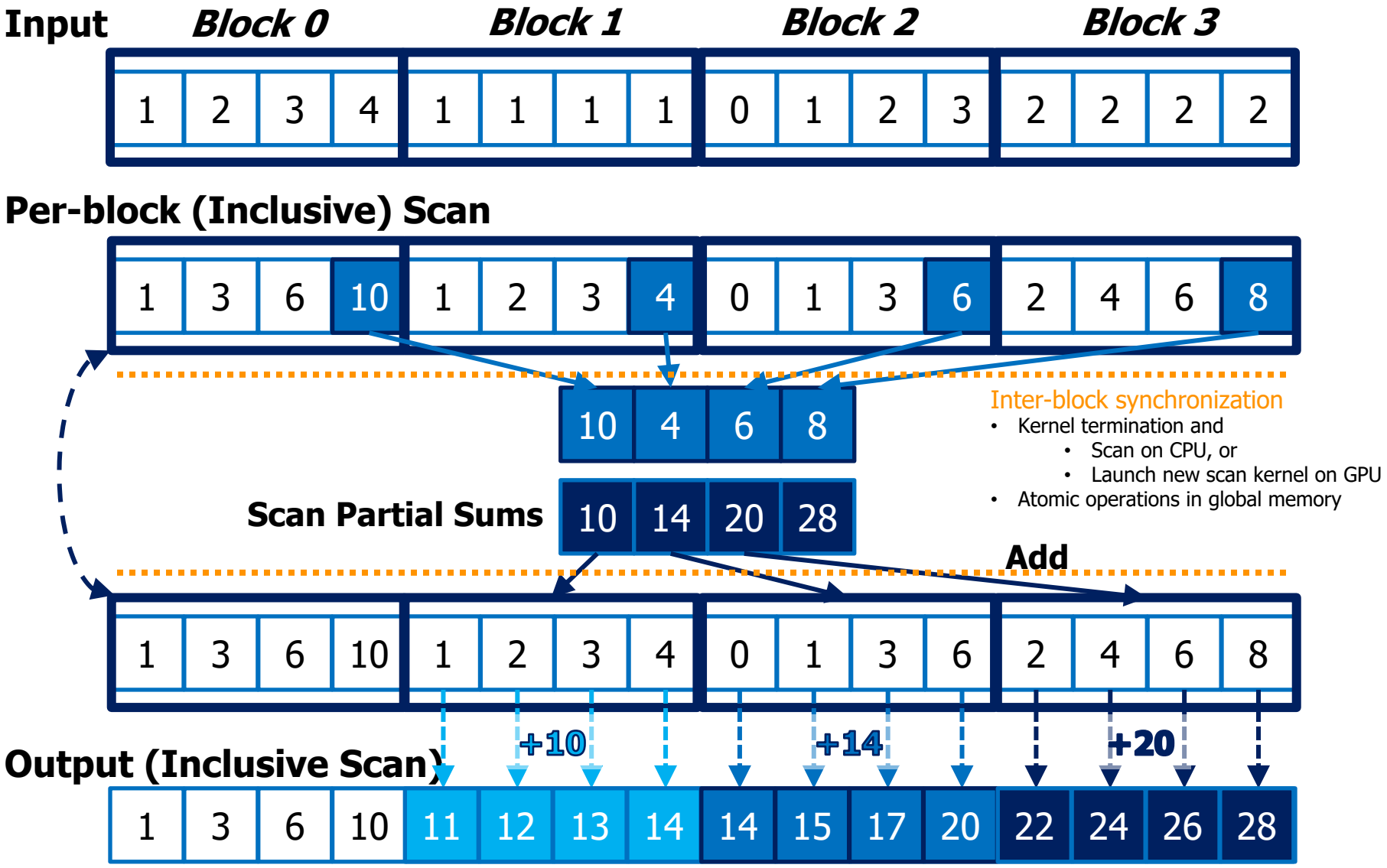
Prefix Sum (Scan)

- Prefix sum or scan is an operation that takes an input array and an associative operator,
  - E.g., addition, multiplication, maximum, minimum
- And returns an output array that is the result of recursively applying the associative operator on the elements of the input array

- Input array $[x_0, x_1, \ldots, x_{n-1}]$
- Associative operator $\oplus$

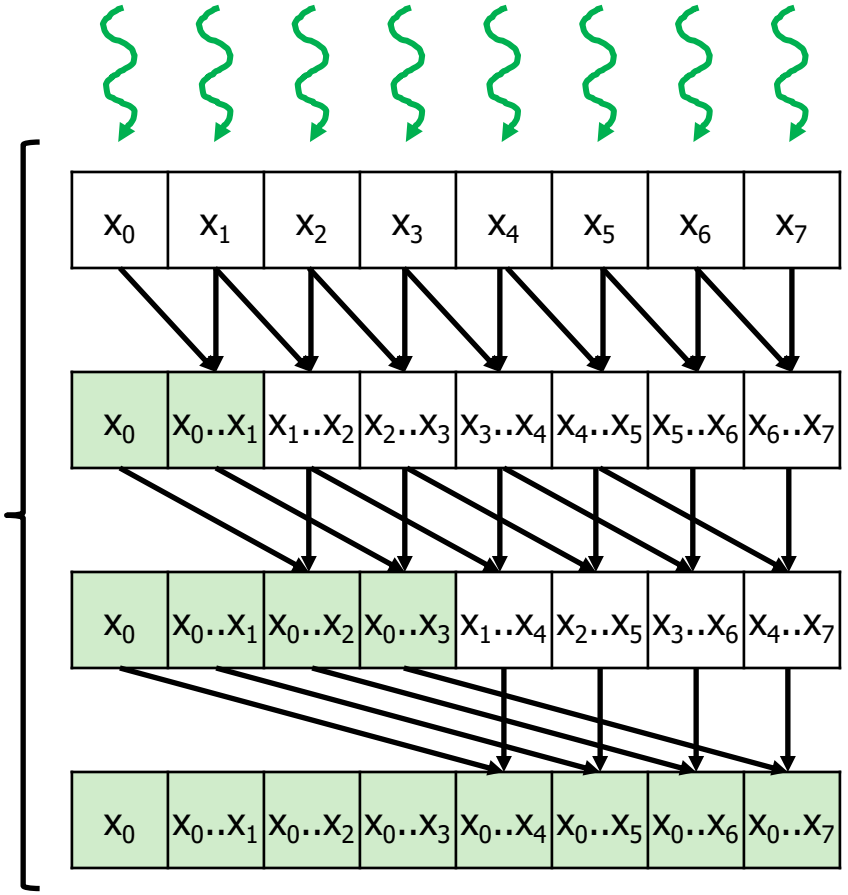
- An output array $[y_0, y_1, \ldots, y_{n-1}]$ where
  - Exclusive scan: $y_i = x_0 \oplus x_1 \oplus \ldots \oplus x_{i-1}$
  - Inclusive scan: $y_i = x_0 \oplus x_1 \oplus \ldots \oplus x_i$

Hierarchical (Inclusive) Scan



Kogge-Stone Parallel (Inclusive) Scan

Observation:
memory locations are reused

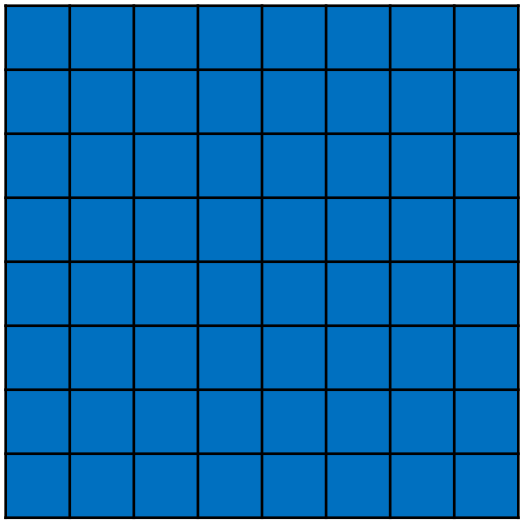


Slide credit: Izzat El Hajj

Sparse Matrices and Sparse Matrix Computation

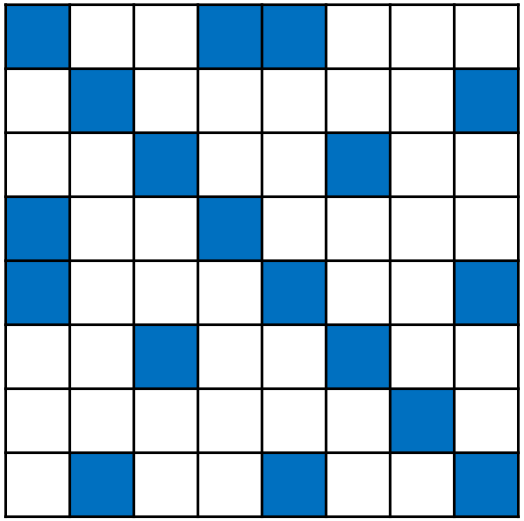
Sparse Matrices

A dense matrix is one where the majority of elements are not zero



A sparse matrix is one where many elements are zero

(many real world systems are sparse)

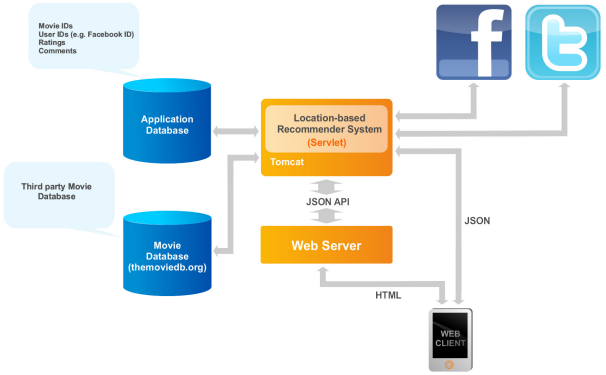


- Opportunities:
  - Do not need to allocate space for zeros (save memory capacity)
  - Do not need to load zeros (save memory bandwidth)
  - Do not need to compute with zeros (save computation time)

Slide credit: Izzat El Hajj

Sparse Matrices are Widespread Today

Recommender Systems



- Collaborative Filtering

Graph Analytics

- PageRank
- Breadth First Search
- Betweenness Centrality

Neural Networks

- Sparse DNNs
- Graph Neural Networks

Real-World Matrices Have High Sparsity

0.0003%

non-zero elements

2.31%

non-zero elements

Sparse matrix compression
is essential to enable
efficient storage and computation

Sparse Matrix Storage Formats

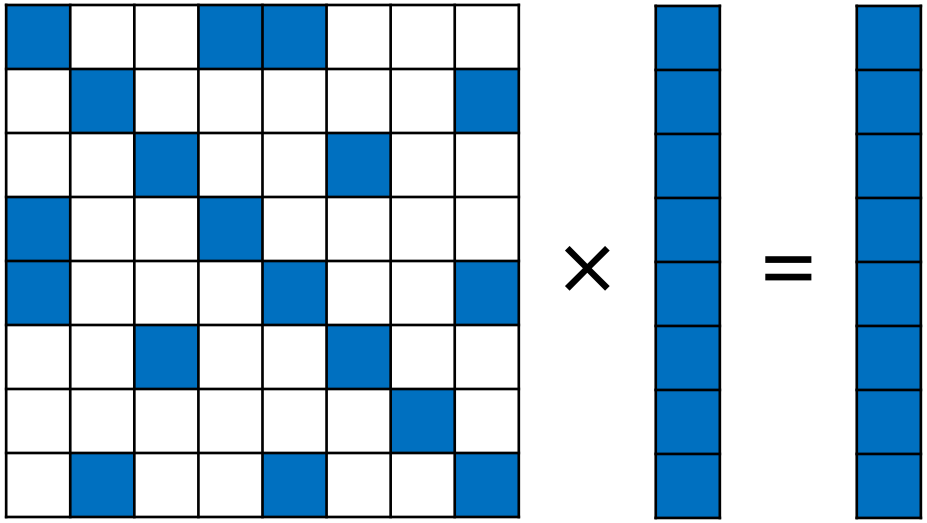
- Many storage formats for sparse matrices:
  - Coordinate Format (COO)
  - Compressed Sparse Row (CSR)
  - ELLPACK Format (ELL)
  - Jagged Diagonal Storage (JDS)
  - …
- Format design considerations:
  - Space efficiency (memory consumed)
  - Flexibility (ease of adding/reordering elements)
  - Accessibility (ease of finding desired data)
  - Memory access pattern (enabling coalescing)
  - Load balance (minimizing control divergence)

Slide credit: Izzat El Hajj

SpMV

- Choice of best format depends on computation and matrix characteristics (e.g., sparsity)

- We will use Sparse Matrix-Vector multiplication (SpMV) as an example to study different formats



Slide credit: Izzat El Hajj

Coordinate Format (COO)

Matrix:

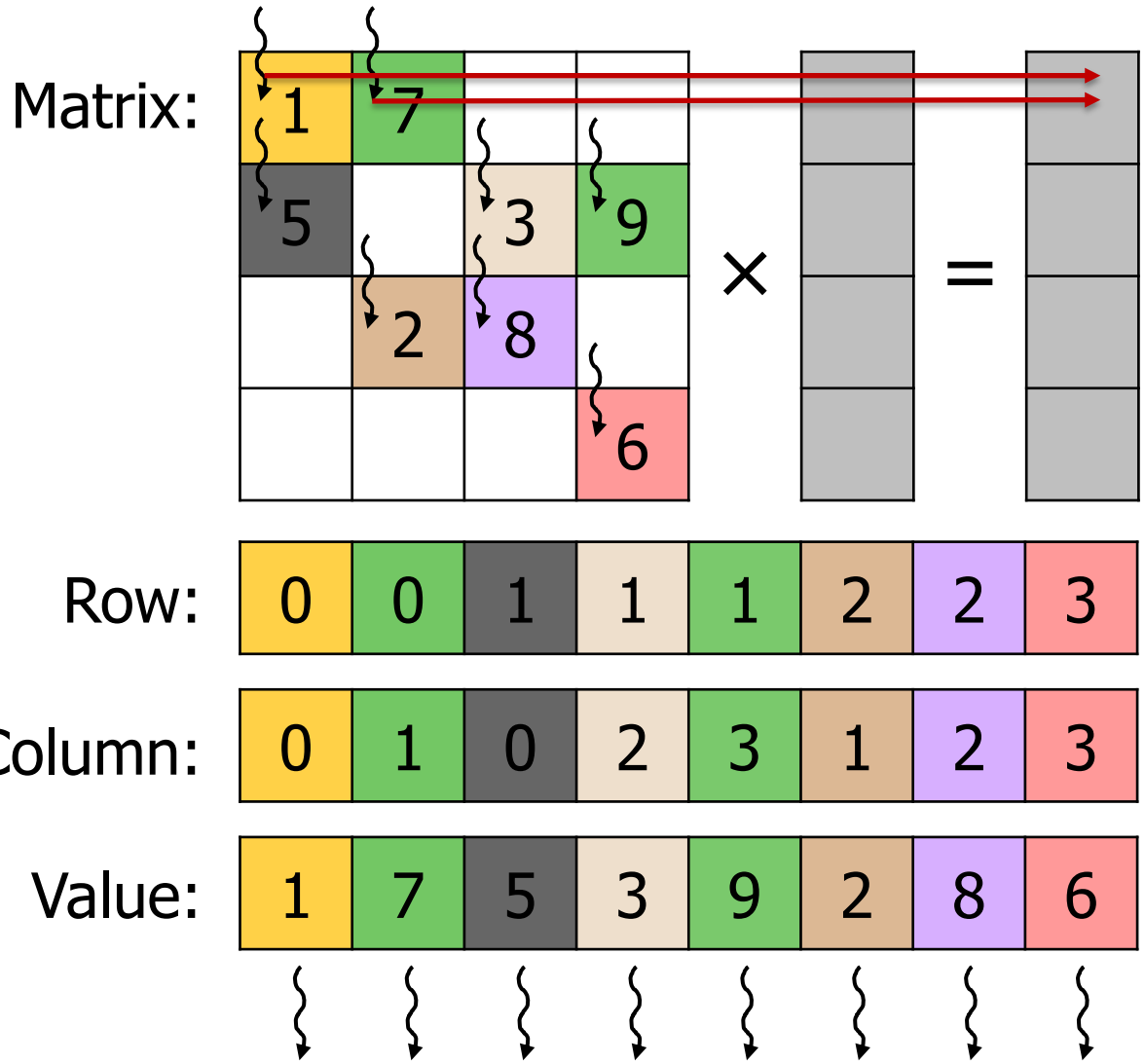
1	7		
5		3	9
	2	8	
			6

Store every nonzero along with its row index and column index

Row:	0	0	1	1	1	2	2	3
Column:	0	1	0	2	3	1	2	3
Value:	1	7	5	3	9	2	8	6

Slide credit: Izzat El Hajj

SpMV/COO



Multiple threads writing to the same output (need atomic operations)

Parallelization approach:
Assign one thread per nonzero

Slide credit: Izzat El Hajj

SpMV/COO Code

```
// All elements of outVector are zero-initialized

__global__ void spmv_coo_kernel(COOMatrix cooMatrix, float* inVector, float* outVector) {

    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

    if(i < cooMatrix.numNonzeros) {

        unsigned int row = cooMatrix.rowIdxs[i]; // Read row and column of element i
        unsigned int col = cooMatrix.colIdxs[i];

        float value = cooMatrix.values[i]; // Read value of element i

        atomicAdd(&outVector[row], inVector[col]*value); // Atomic addition
    }
}
```


Slide credit: Izzat El Hajj

Recall: Coalesced Atomic Operations

- Identify threads operating on the same atomic and use a reduction

```
int atomic_add(int * ptr, int value){

   unsigned active_mask = __activemask();
   unsigned active_mask = __match_any_sync(active_mask, ptr);

   int value = reduce_warp(active_mask, value);

   if((__ffs(active_mask) - 1) == lane) {

      value = atomicAdd(ptr, value);
   }

   value = __shfl_sync(active_mask, value, __ffs(active_mask) - 1);
   return value;
}
```


Slide credit: Hwu & Kirk (PUMPS 2021)

COO Tradeoffs

- Advantages:
  - Flexibility: easy to add new elements to the matrix, nonzeros can be stored in any order
  - Accessibility: given nonzero, easy to find row and column
  - SpMV/COO has coalesced memory accesses
  - SpMV/COO has no control divergence

- Disadvantages:
  - Accessibility: given a row or a column, hard to find all nonzeros (need to search)
  - SpMV/COO uses atomic operations

Slide credit: Izzat El Hajj

Widely Used Format: Compressed Sparse Row

- Compressed Sparse Row (CSR) provides high compression ratio
- Used in multiple libraries & frameworks:
  - Intel MKL[1]
  - TACO[2]
  - Ligra[3]
  - Polymer[4]
  - Gunrock[5]
  - Etc.

[1] Intel Math Kernel Library, http://software.intel.com/en-us/articles/intel-mkl/

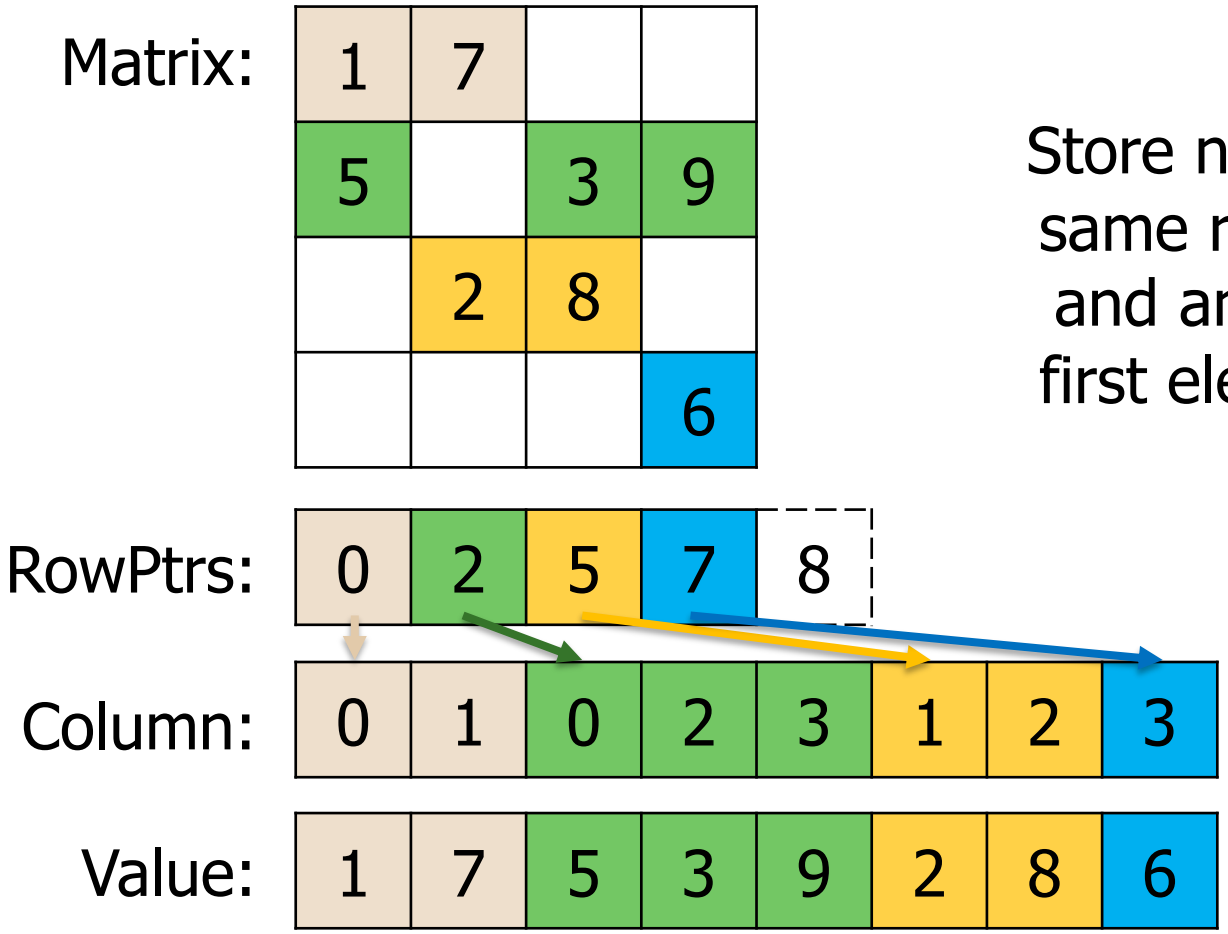
[2] F. Kjolstad, S. Chou, D. Lugato, S. Kamil, and S. Amarasinghe, "taco: A Tool to Generate Tensor Algebra Kernels," ASE 2017

[3] J. Shun and G. E. Blelloch, "Ligra: A Lightweight Graph Processing Framework for Shared Memory," PPoPP 2013

[4] K. Zhang, R. Chen, and H. Chen, "Numa-aware Graph-structured Analytics," PPoPP 2015

[5] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A High-performance Graph Processing Library on the GPU," PPoPP 2016

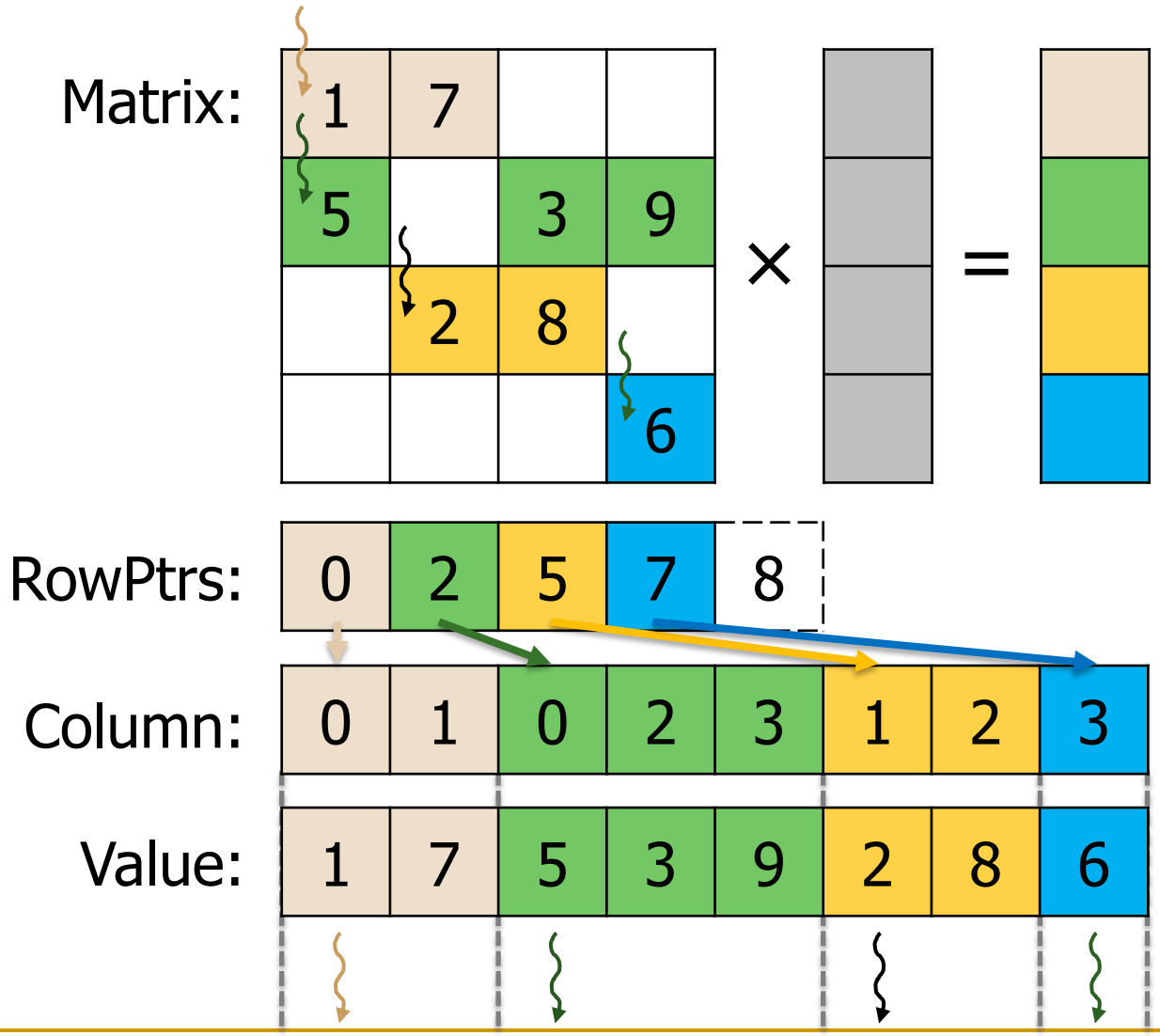
Compressed Sparse Row (CSR)



Store nonzeros of the same row adjacently and an index to the first element of each row

Slide credit: Izzat El Hajj

SpMV/CSR

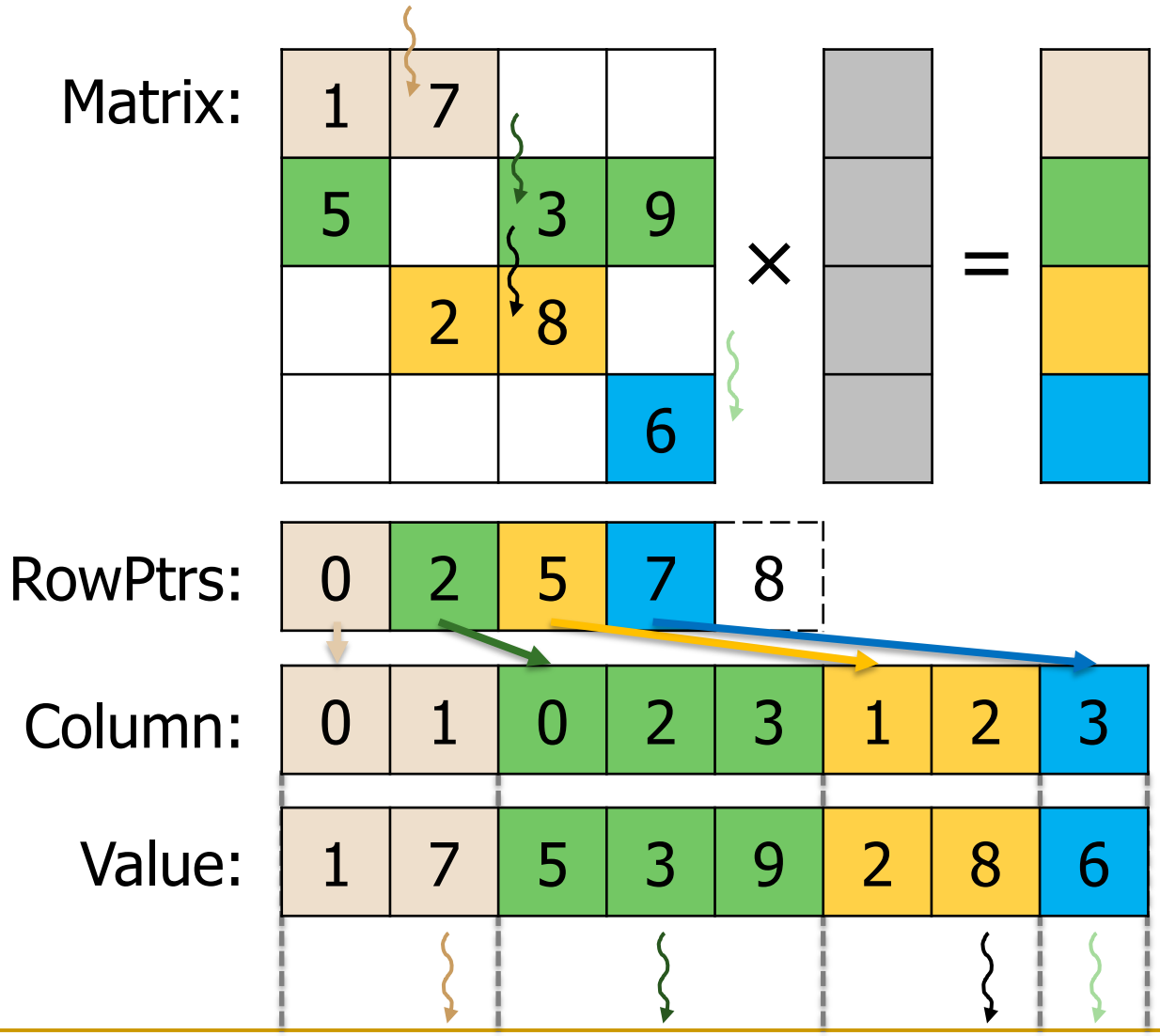


Parallelization approach:

Assign one thread to loop over each input row sequentially and update corresponding output element

Slide credit: Izzat El Hajj

SpMV/CSR

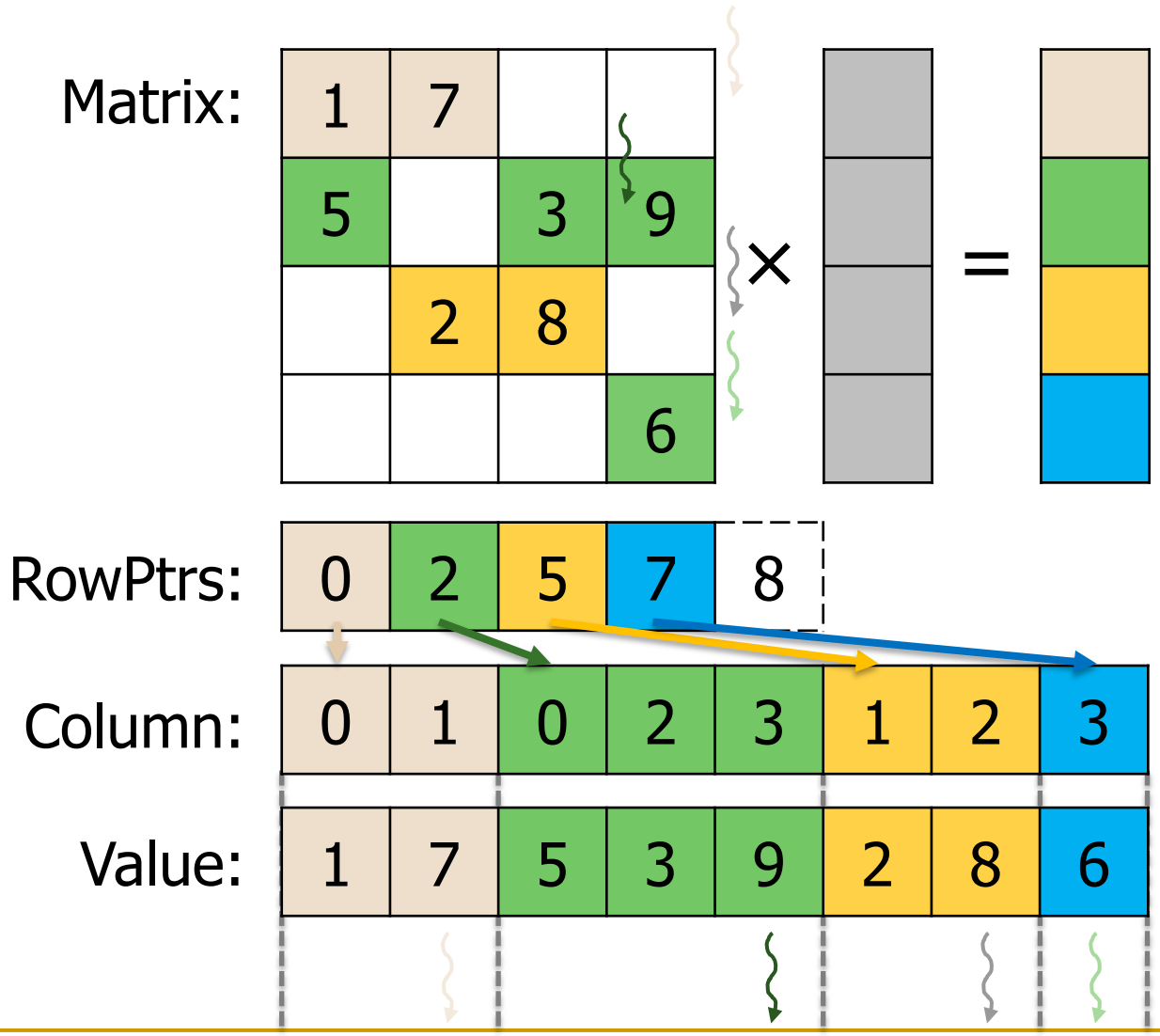


Parallelization approach:

Assign one thread to loop over each input row sequentially and update corresponding output element

Slide credit: Izzat El Hajj

SpMV/CSR



Parallelization approach:

Assign one thread to loop over each input row sequentially and update corresponding output element

Slide credit: Izzat El Hajj

SpMV/CSR Code

```
__global__ void spmv_csr_kernel(CSRMatrix csrMatrix, float* inVector, float* outVector) {

    unsigned int row = blockIdx.x*blockDim.x + threadIdx.x; // Each thread works on one row

    if(row < csrMatrix.numRows) {
        float sum = 0.0f; // Zero initialize temporal accumulator

        // Loop over all elements of a row
        for(unsigned int i = csrMatrix.rowPtrs[row]; i < csrMatrix.rowPtrs[row + 1]; ++i) {

            unsigned int col = csrMatrix.colIdxs[i]; // Read column index

            float value = csrMatrix.values[i]; // Read value

            sum += inVector[col]*value; // Multiply and accumulate
        }

        outVector[row] = sum;
    }
}
```


Slide credit: Izzat El Hajj

CSR Tradeoffs (versus COO)

- Advantages:
  - Space efficiency: row pointers smaller than row indexes
  - Accessibility: given a row, easy to find all nonzeros
  - SpMV/CSR avoids atomics, every thread owns its output
- Disadvantages:
  - Flexibility: hard to add new elements to the matrix
  - Accessibility: given nonzero, hard to find row; given a column, hard to find all nonzeros
  - SpMV/CSR memory accesses are not coalesced
  - SpMV/CSR has control divergence

Slide credit: Izzat El Hajj

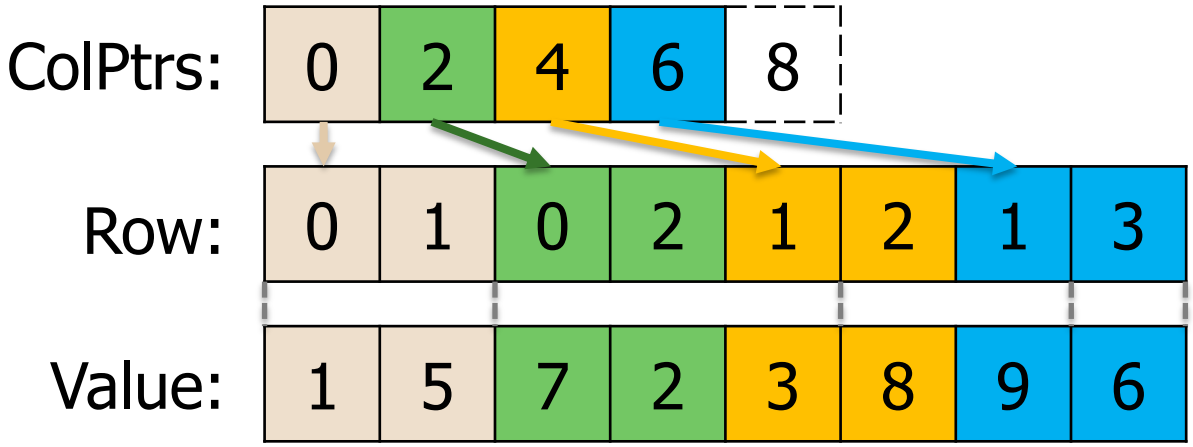
Compressed Sparse Column (CSC)

Matrix:

1	7		
5		3	9
	2	8	
			6

Like CSR, but groups nonzeros by column

(useful for computations other than SpMV that require column traversal)



Slide credit: Izzat El Hajj

ELLPACK Format (ELL)

Matrix:

1	7		
5		3	9
	2	8	
			6

Group nonzeros by row (like CSR)…

Column:

0	1	
0	2	3
1	2	
3		

Value:

1	7	
5	3	9
2	8	
6		

Slide credit: Izzat El Hajj

ELLPACK Format (ELL)

Matrix:

1	7		
5		3	9
	2	8	
			6

…pad rows so they all have the same size…

Column:

0	1	*
0	2	3
1	2	*
3	*	*

Value:

1	7	*
5	3	9
2	8	*
6	*	*

Slide credit: Izzat El Hajj

ELLPACK Format (ELL)

Matrix:

1	7		
5		3	9
	2	8	
			6

…and store padded array of nonzeros in column major order

Column:

0	1	*
0	2	3
1	2	*
3	*	*

Value:

1	7	*
5	3	9
2	8	*
6	*	*

Slide credit: Izzat El Hajj

ELLPACK Format (ELL)

Matrix:

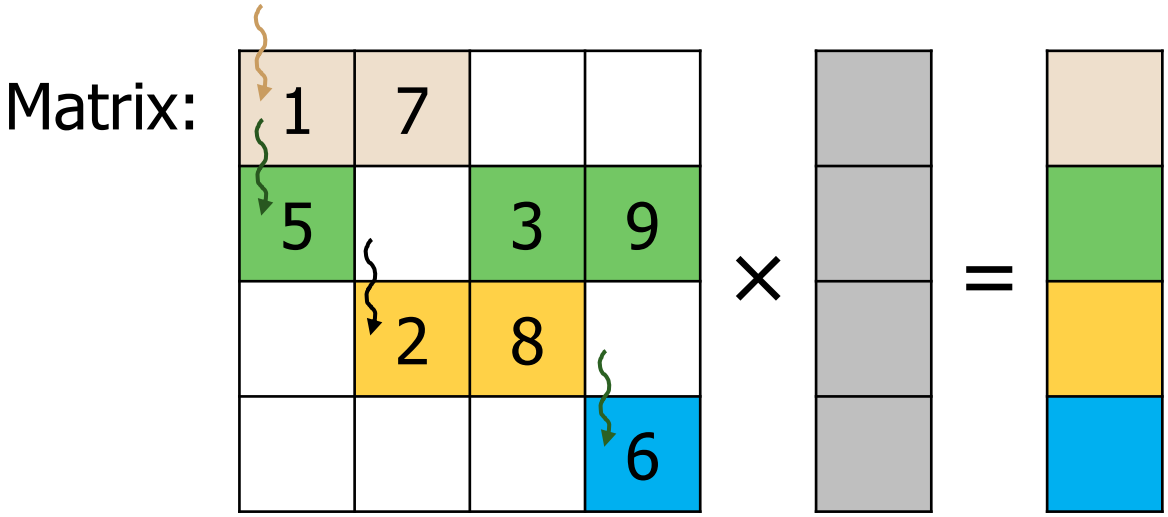
1	7		
5		3	9
	2	8	
			6

…and store padded array of nonzeros in column major order

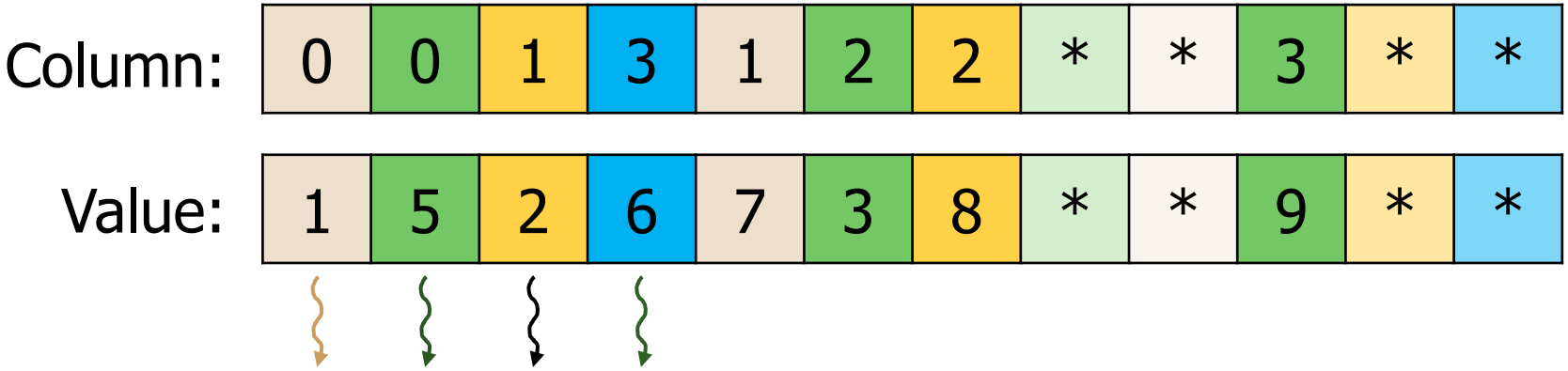
Column:	0	0	1	3	1	2	2	*	*	3	*	*
Value:	1	5	2	6	7	3	8	*	*	9	*	*

Slide credit: Izzat El Hajj

SpMV/ELL



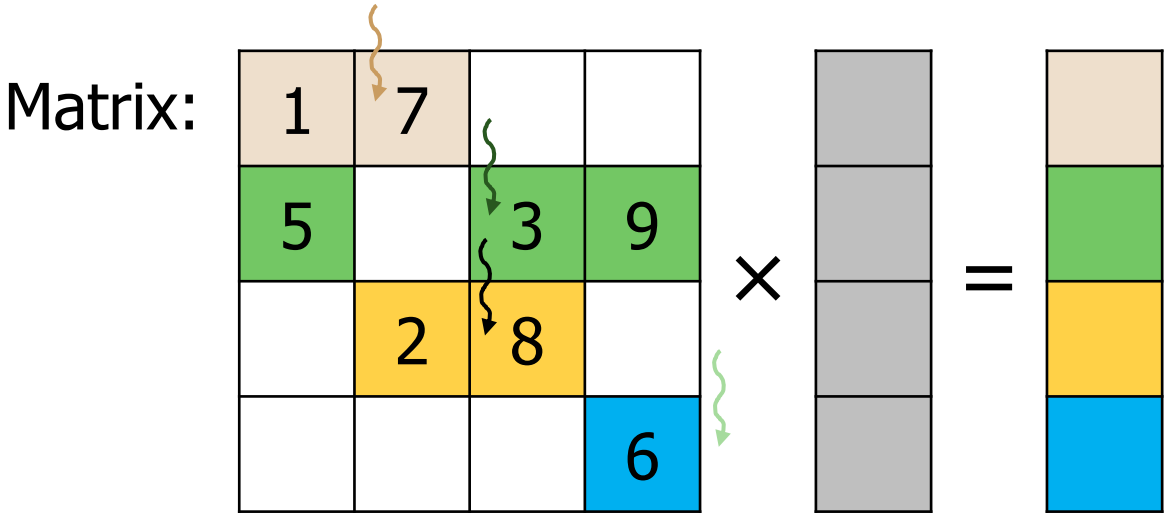
Parallelization approach:
Assign one thread to loop over each input row sequentially and update corresponding output element



Memory accesses are coalesced

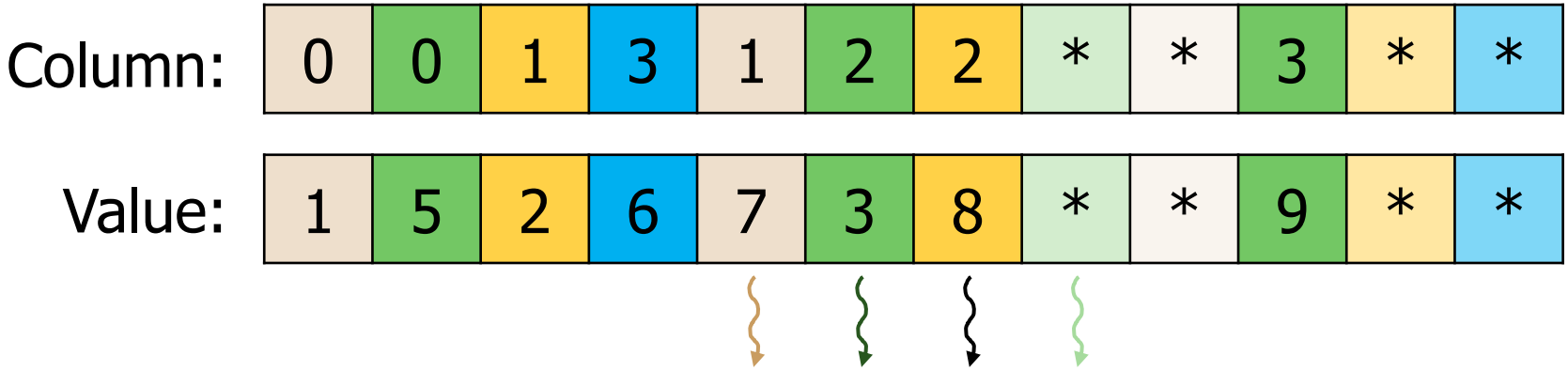
Slide credit: Izzat El Hajj

SpMV/ELL



Parallelization approach:

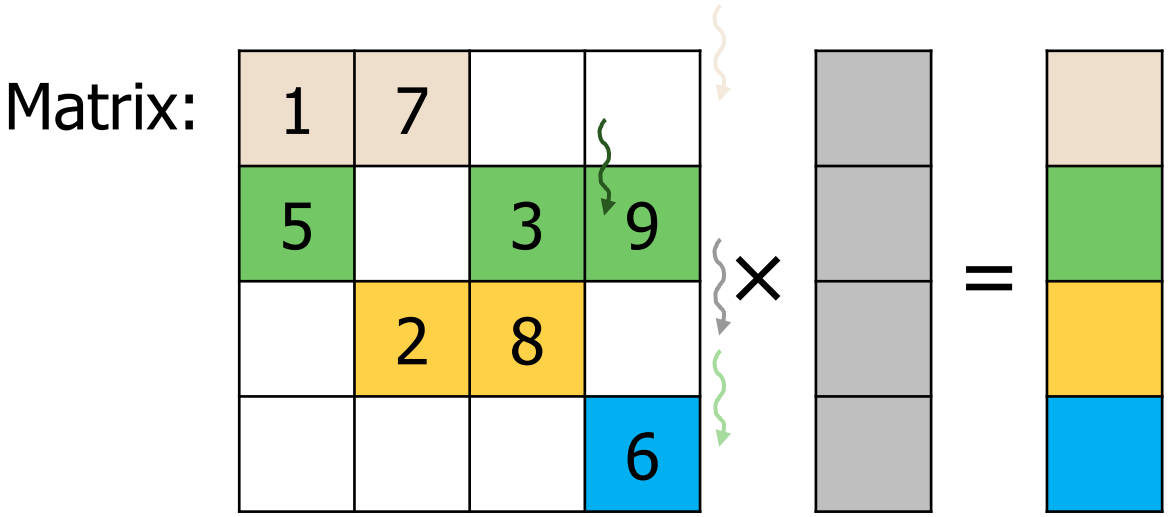
Assign one thread to loop over each input row sequentially and update corresponding output element



Memory accesses are coalesced

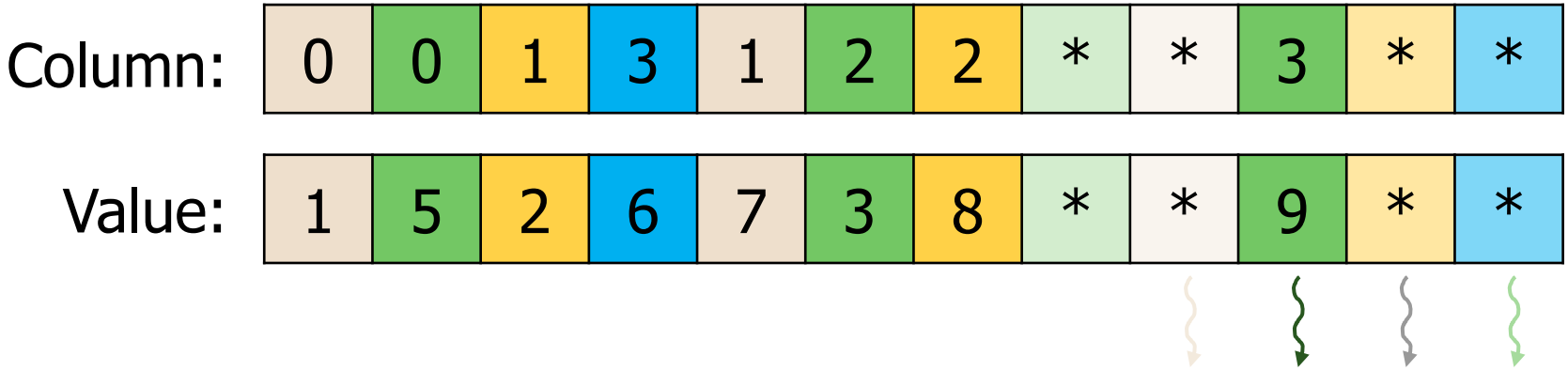
Slide credit: Izzat El Hajj

SpMV/ELL



Parallelization approach:

Assign one thread to loop over each input row sequentially and update corresponding output element



Memory accesses are coalesced

Slide credit: Izzat El Hajj

SpMV/ELL Code

```
__global__ void spmv_ell_kernel(ELLMatrix ellMatrix, float* inVector, float* outVector) {

    unsigned int row = blockIdx.x*blockDim.x + threadIdx.x; // Each thread works on one row

    if(row < ellMatrix.numRows) {
        float sum = 0.0f; // Zero initialize temporal accumulator

        // Loop over all non-zero elements in the own row
        for(unsigned int nnzIdx = 0; nnzIdx < ellMatrix.nnzPerRow[row]; ++nnzIdx) {

            unsigned int i = nnzIdx*ellMatrix.numRows + row; // Index of non-zero element

            unsigned int col = ellMatrix.colIdxs[i]; // Read column index

            float value = ellMatrix.values[i]; // Read value

            sum += inVector[col]*value; // Multiply and accumulate
        }

        outVector[row] = sum;
    }
}
```


Slide credit: Izzat El Hajj

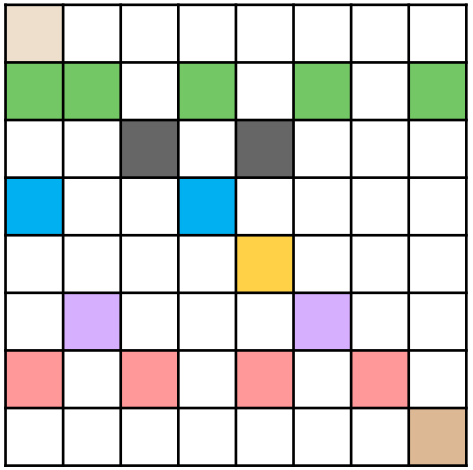
ELL Tradeoffs

- Advantages:
  - Flexibility: can add new elements as long as row not full
  - Accessibility: given a row, easy to find all nonzeros; given nonzero, easy to find row and column
  - SpMV/ELL memory accesses are coalesced
- Disadvantages:
  - Space efficiency: overhead due to padding
  - Accessibility: given a column, hard to find all nonzeros
  - SpMV/ELL has control divergence

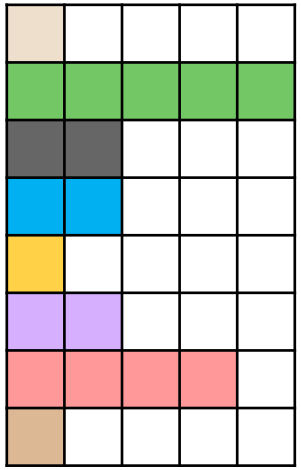
Slide credit: Izzat El Hajj

Hybrid ELL + COO

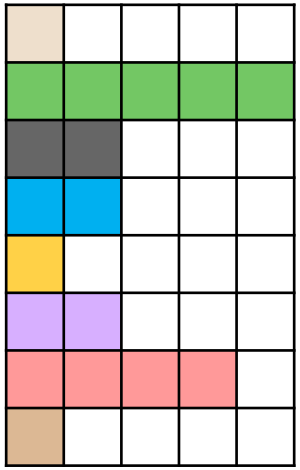
Matrix:



Column:

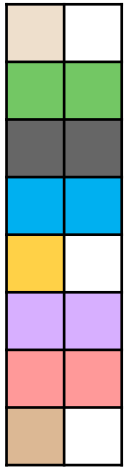


Value:

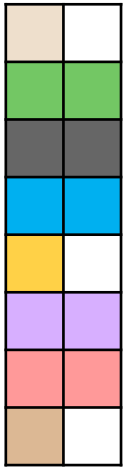


ELL Format

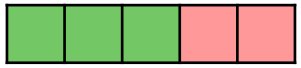
$Column_{ELL}$:



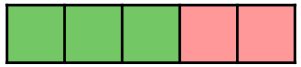
$Value_{ELL}$:



Row_{COO}:



$Column_{COO}$:



$Value_{COO}$:

Hybrid ELL + COO

(use COO for very long rows)

Slide credit: Izzat El Hajj

ELL + COO Tradeoffs

- Similar to ELL, with the following added benefits from using COO:
  - Space efficiency: less padding
  - Flexibility: can add new elements to any row
  - Less control divergence

Slide credit: Izzat El Hajj

Jagged Diagonal Storage (JDS)

Matrix:

a					
b	c		d	e	
		f		g	
h			i		j
	k			l	
m		n			o

Group nonzeros by row (like CSR)…

Column:

0			
0	1	3	4
2	4		
0	3	5	
1	4		
0	2	5	

Value:

a			
b	c	d	e
f	g		
h	i	j	
k	l		
m	n	o	

Slide credit: Izzat El Hajj

Jagged Diagonal Storage (JDS)

Matrix:

a					
b	c		d	e	
		f		g	
h			i		j
	k			l	
m		n			o

...sort rows by size and remember the original row index...

Column:

0	1	3	4
0	3	5	
0	2	5	
2	4		
1	4		
0			

Value:

b	c	d	e
h	i	j	
m	n	o	
f	g		
k	l		
a			

Row:

1
3
5
2
4
0

Slide credit: Izzat El Hajj

Jagged Diagonal Storage (JDS)

Matrix:

a					
b	c		d	e	
		f		g	
h			i		j
	k			l	
m		n			o

...store nonzeros in column major...

Column:

0	1	3	4
0	3	5	
0	2	5	
2	4		
1	4		
0			

Value:

b	c	d	e
h		i	
m	n	o	
f	g		
k			
a			

Row:

1
3
5
2
4
0

Slide credit: Izzat El Hajj

Jagged Diagonal Storage (JDS)

Matrix:

a					
b	c		d	e	
		f		g	
h			i		j
	k			l	
m		n			o

...store nonzeros in column major...

Row:

1
3
5
2
4
0

Column:	0	0	0	2	1	0	1	3	2	4	4	3	5	5	4
Value:	b	h	m	f	k	a	c	i	n	g	l	d	j	o	e

Slide credit: Izzat El Hajj

Jagged Diagonal Storage (JDS)

Matrix:

a					
b	c		d	e	
		f		g	
h			i		j
	k			l	
m		n			o

…and remember where the nonzeros of each iteration start

IterPtr:

0	6	11	14	15

Column:

0	0	0	2	1	0	1	3	2	4	4	3	5	5	4

Value:

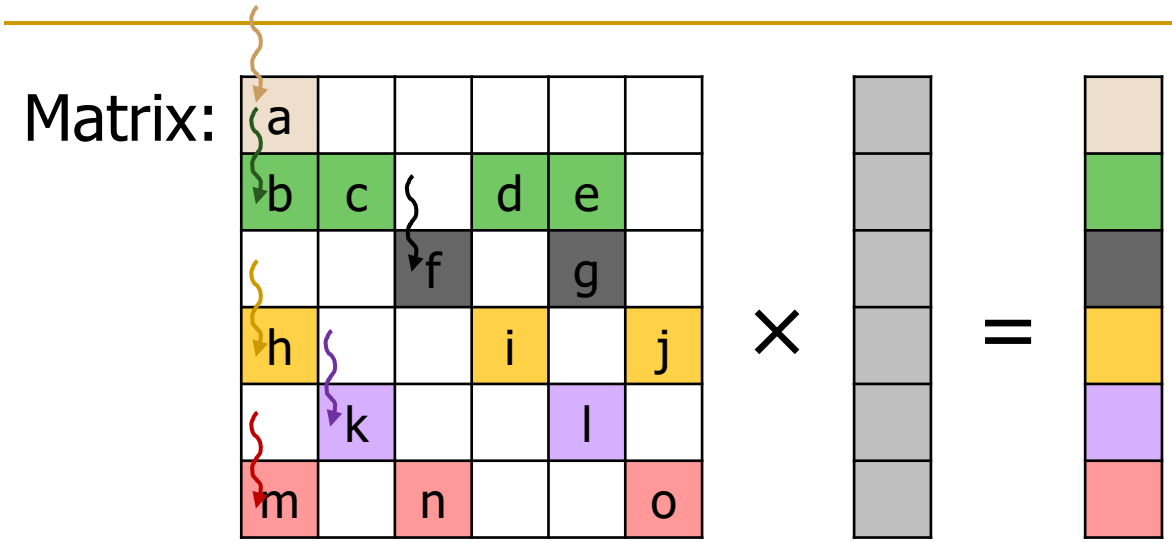
b	h	m	f	k	a	c	i	n	g	l	d	j	o	e

Row:

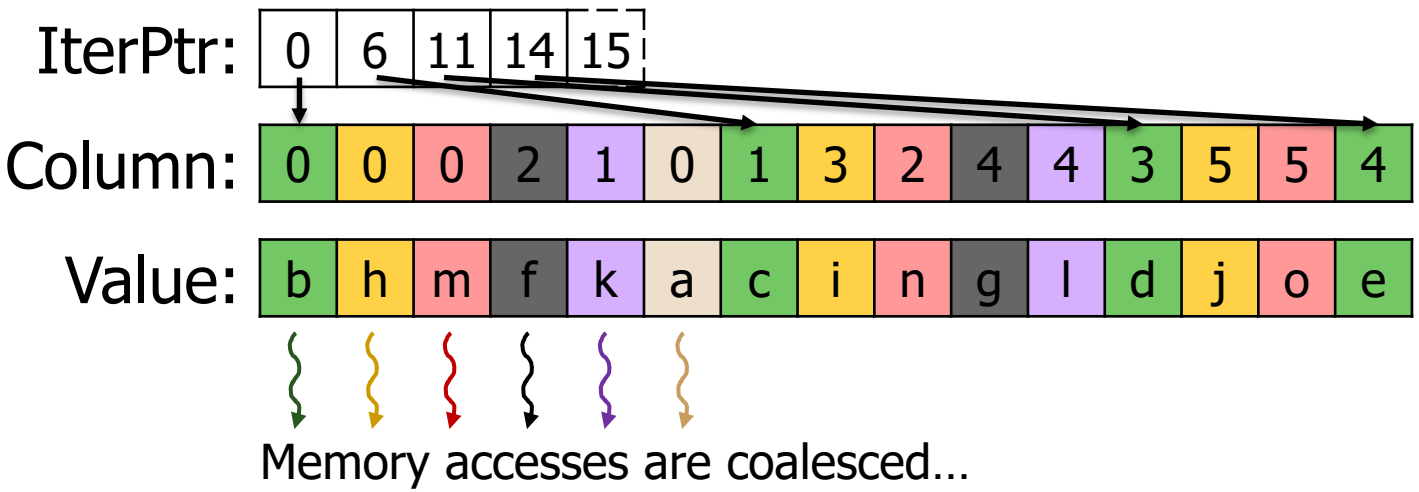
1
3
5
2
4
0

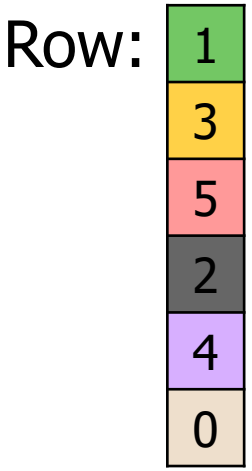
Slide credit: Izzat El Hajj

SpMV/JDS Kernel



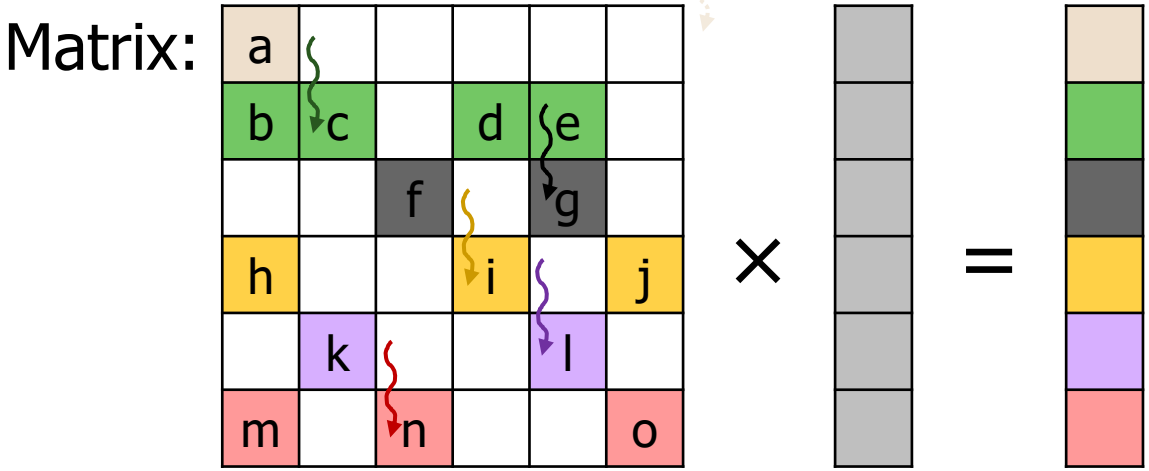
Parallelization approach:
Assign one thread to loop over each input row sequentially and update corresponding output element





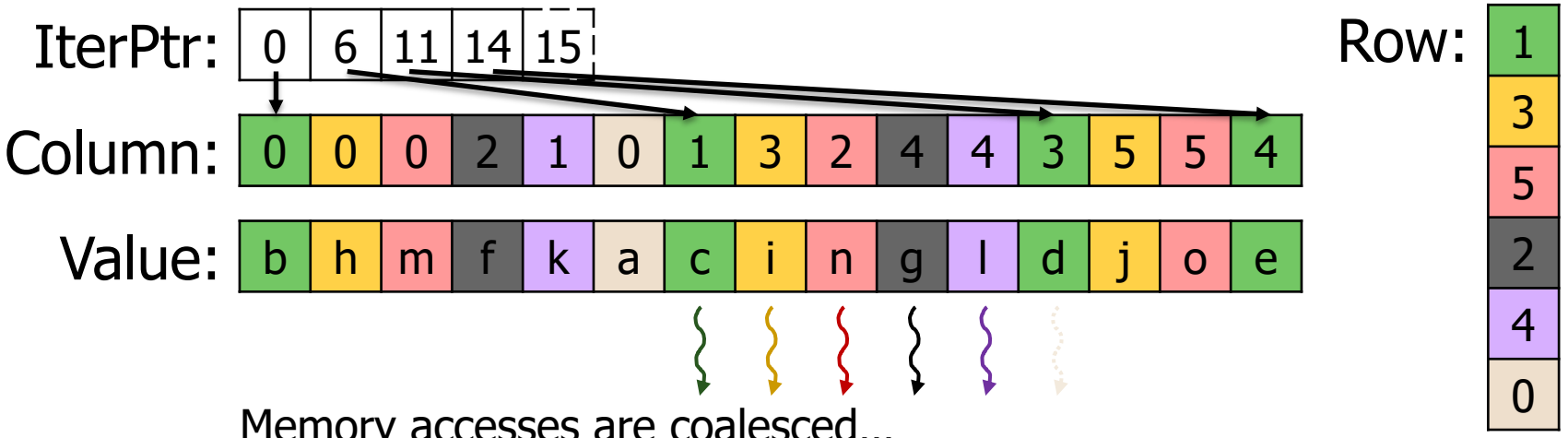
Slide credit: Izzat El Hajj

SpMV/JDS Kernel



Parallelization approach:

Assign one thread to loop over each input row sequentially and update corresponding output element

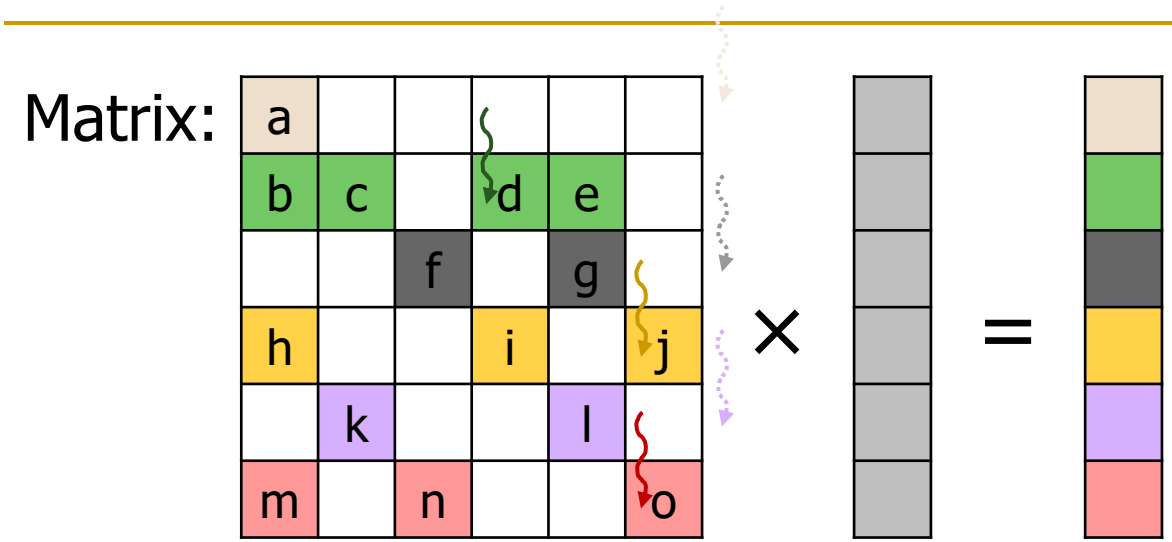


Memory accesses are coalesced…

…and threads drop out from the end, minimizing control divergence

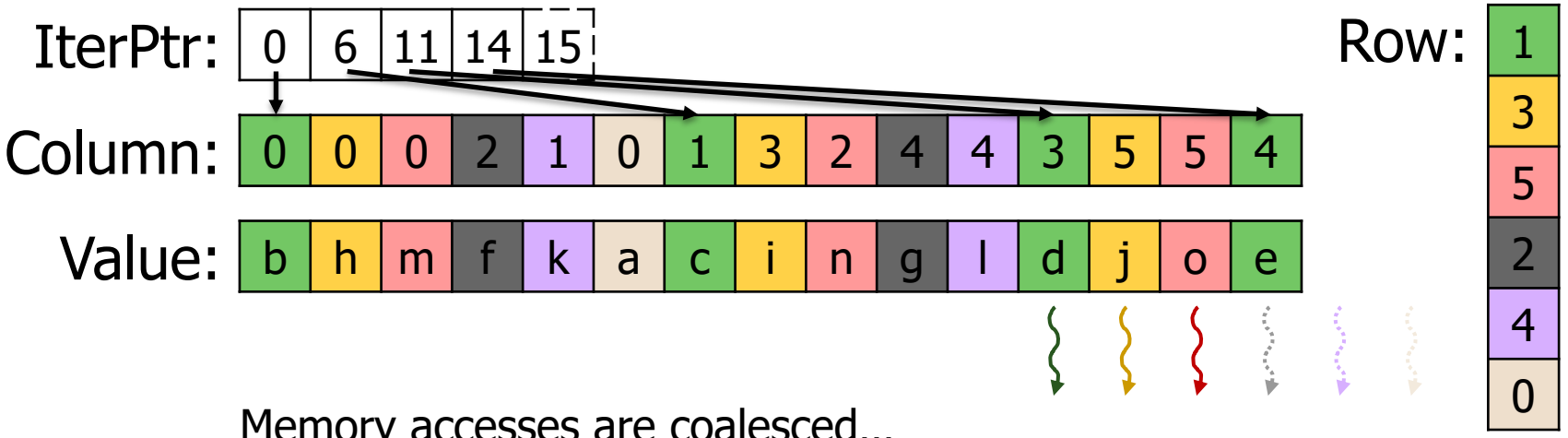
Slide credit: Izzat El Hajj

SpMV/JDS Kernel



Parallelization approach:

Assign one thread to loop over each input row sequentially and update corresponding output element

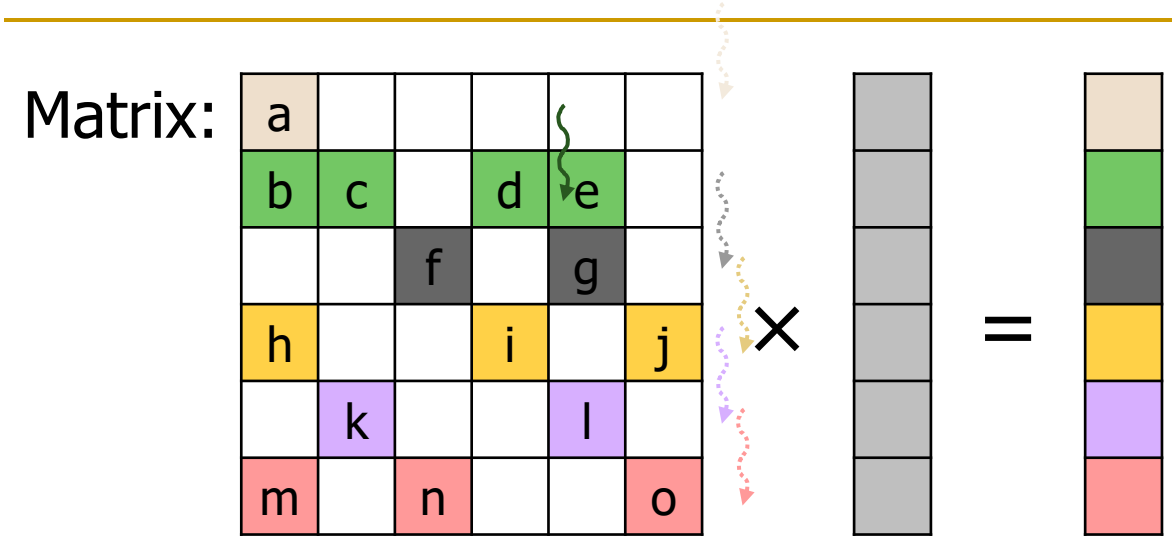


Memory accesses are coalesced…

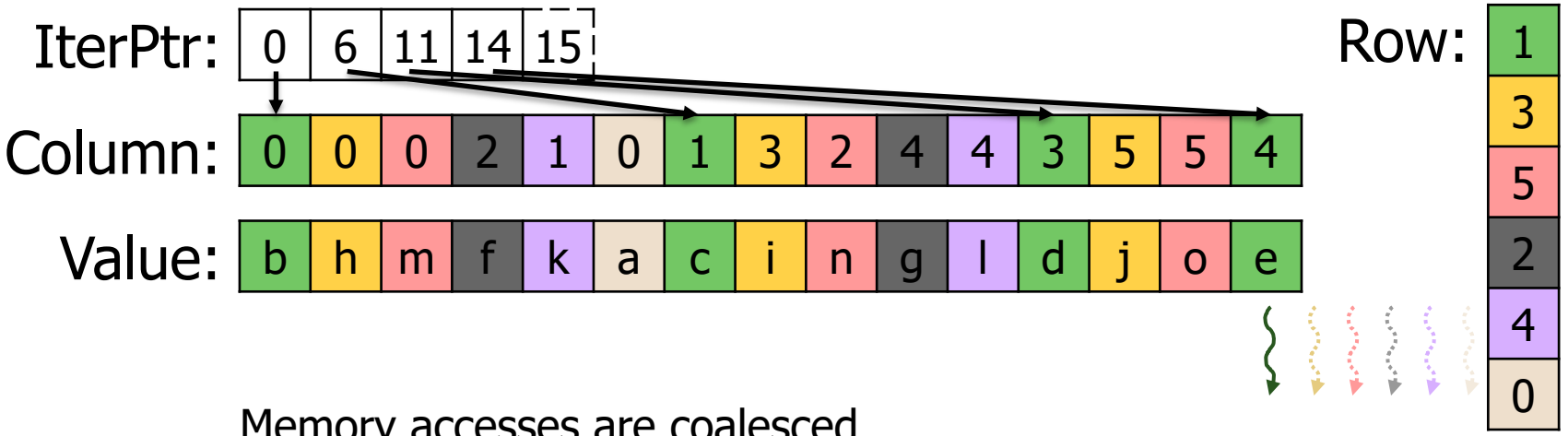
…and threads drop out from the end, minimizing control divergence

Slide credit: Izzat El Hajj

SpMV/JDS Kernel



Parallelization approach:
Assign one thread to loop over each input row sequentially and update corresponding output element



Memory accesses are coalesced…
…and threads drop out from the end, minimizing control divergence

Slide credit: Izzat El Hajj

JDS Tradeoffs

- Advantages:
  - Space efficiency: no padding
  - Accessibility: given a row, easy to find all nonzeros
  - SpMV/JDS memory accesses are coalesced
  - SpMV/JDS minimizes control divergence

- Disadvantages:
  - Flexibility: hard to add new elements to the matrix
  - Accessibility: given nonzero, hard to find row; given a column, hard to find all nonzeros

Slide credit: Izzat El Hajj

Can We Design a Better Compression Format?

Indexing Overhead in Sparse Kernels

Sparse Matrix Vector Multiplication (SpMV)

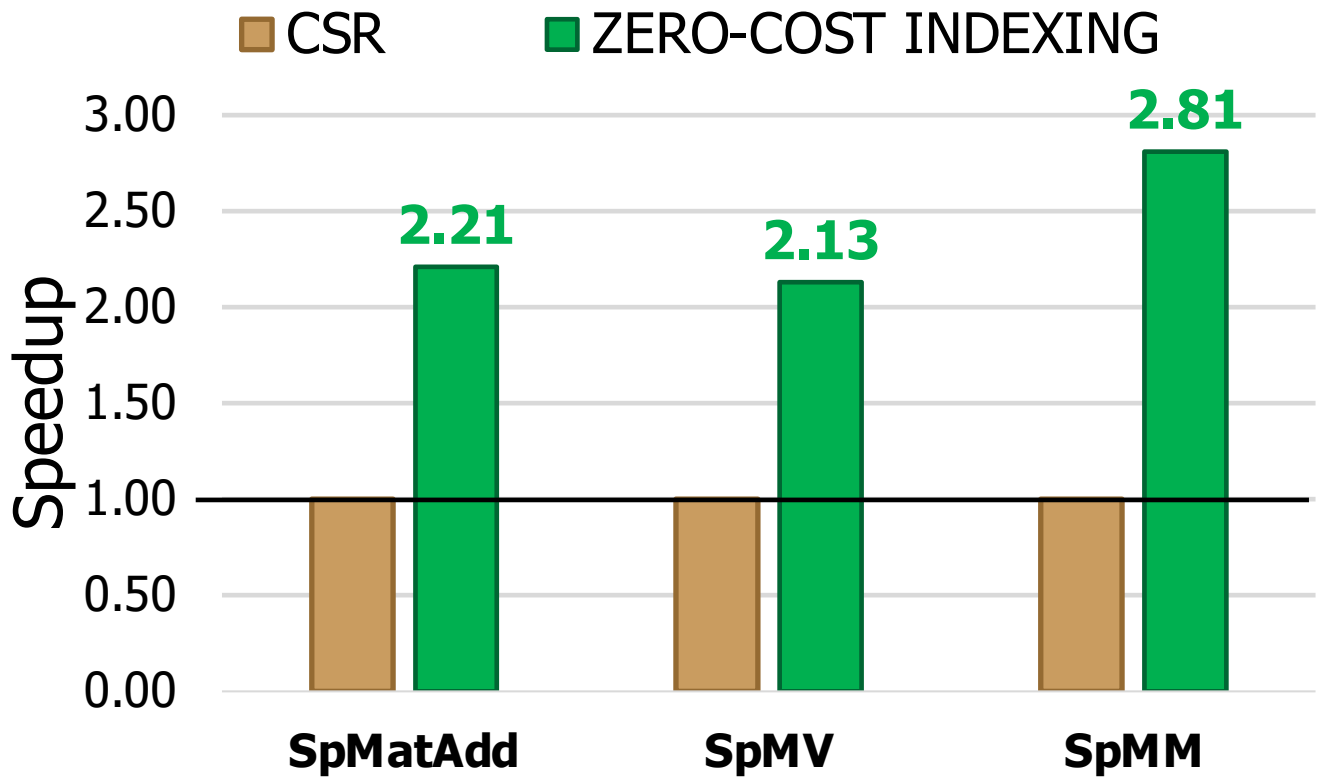
Indexing for every non-zero element of the sparse matrix to multiply with the corresponding element of the vector

Sparse Matrix Matrix Multiplication (SpMM)

Index matching for every inner product between the 2 sparse matrices

Indexing is expensive
for major sparse matrix kernels

Performing Indexing with Zero Cost



Reducing the cost of indexing
can accelerate sparse matrix operations

Limitations of Existing Compression Formats

❶

General formats optimize for storage → Expensive discovery of the positions of non-zero elements

❷

Specialized formats assume specific matrix structures and patterns (e.g., diagonals) → Narrow applicability

What is an Ideal Compression Format?

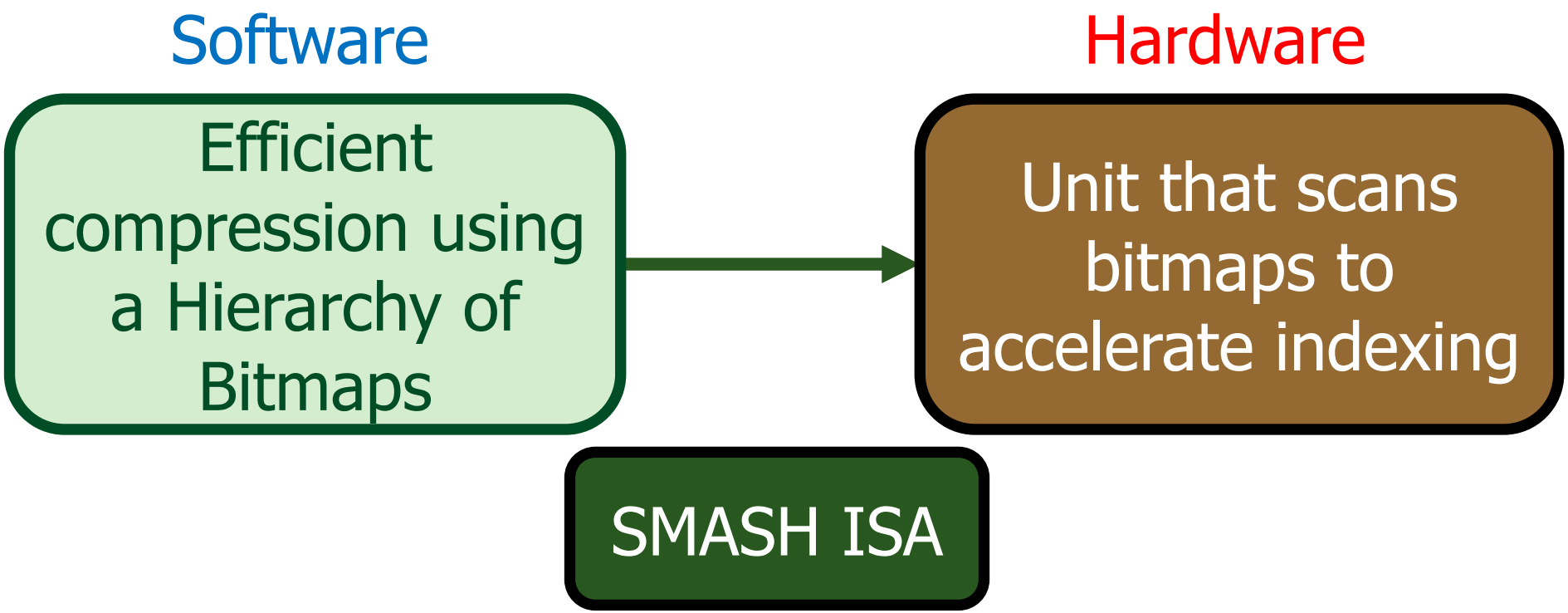
- A sparse matrix compression mechanism that:
  - Minimizes the indexing overheads
  - Can be used across a wide range of sparse matrices and sparse matrix operations
  - Enables high compression ratio
- SMASH* proposes a bitmap-based compression format

* Kanellopoulos et al., "SMASH: Co-designing Software Compression and Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations," MICRO 2019

SMASH: Key Idea

Hardware/Software cooperative mechanism:

- Enables highly-efficient sparse matrix compression and computation
- General across a diverse set of sparse matrices and sparse matrix operations



SMASH: Software Compression Scheme (I)

Software

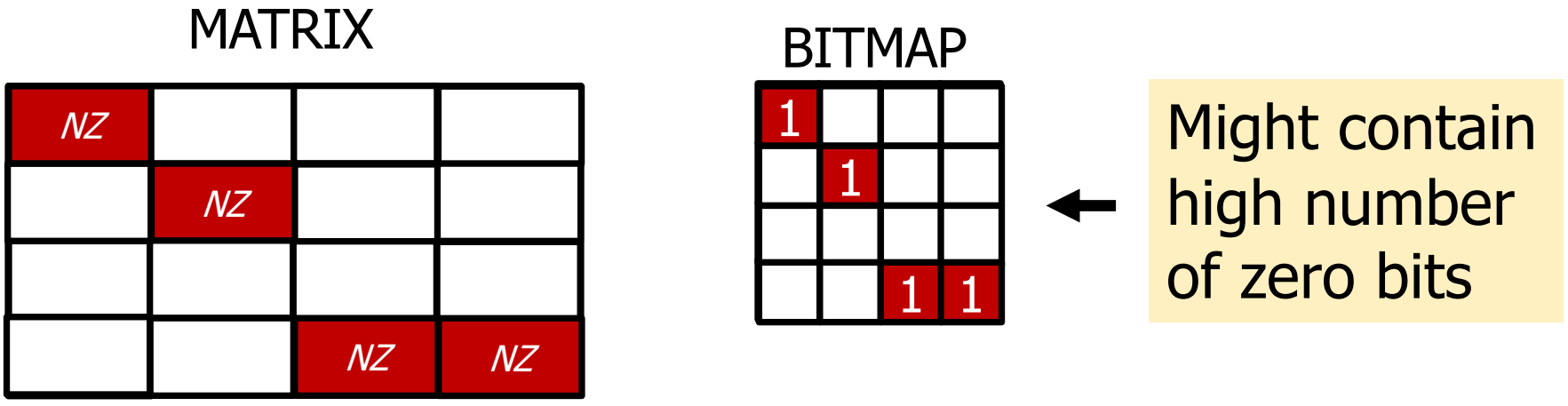
Efficient compression using a Hierarchy of Bitmaps

- Encodes the positions of non-zero elements using bits to maintain low storage overhead

SMASH: Software Compression Scheme (II)

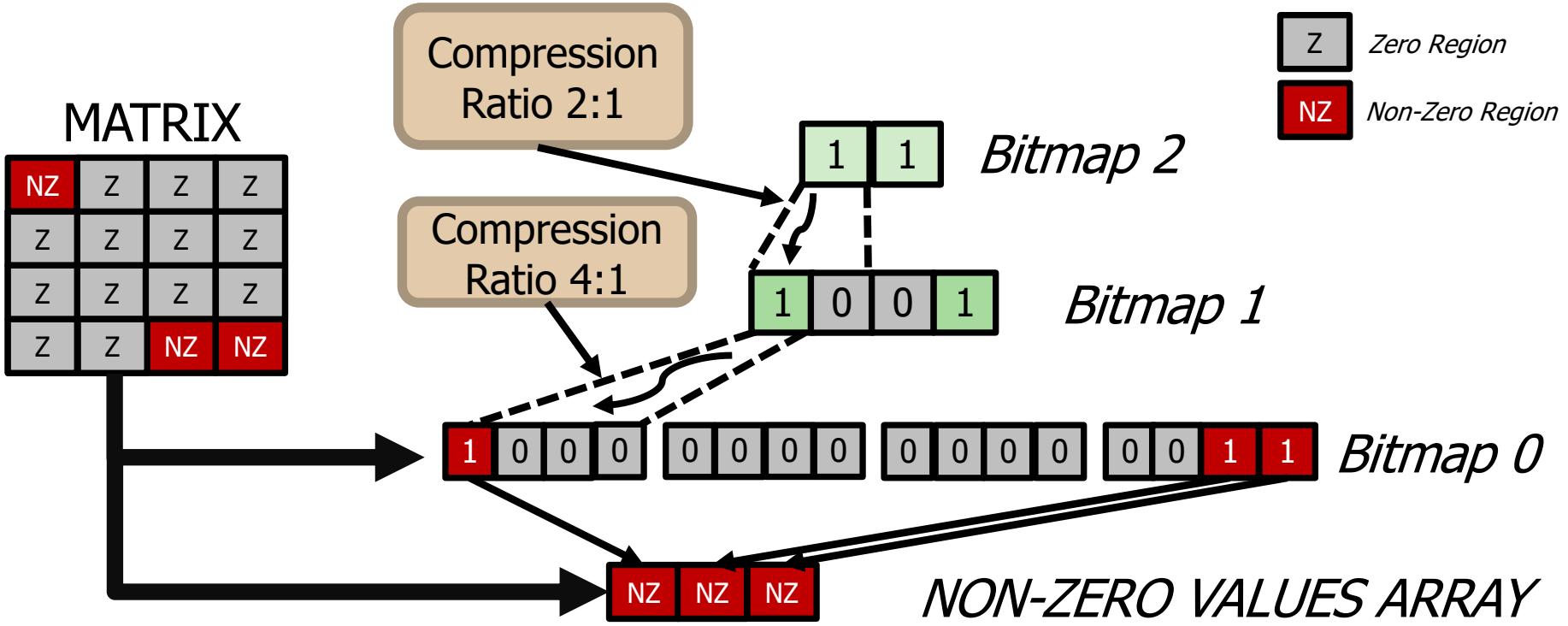
BITMAP:

Encodes if a block of the matrix contains any non-zero element



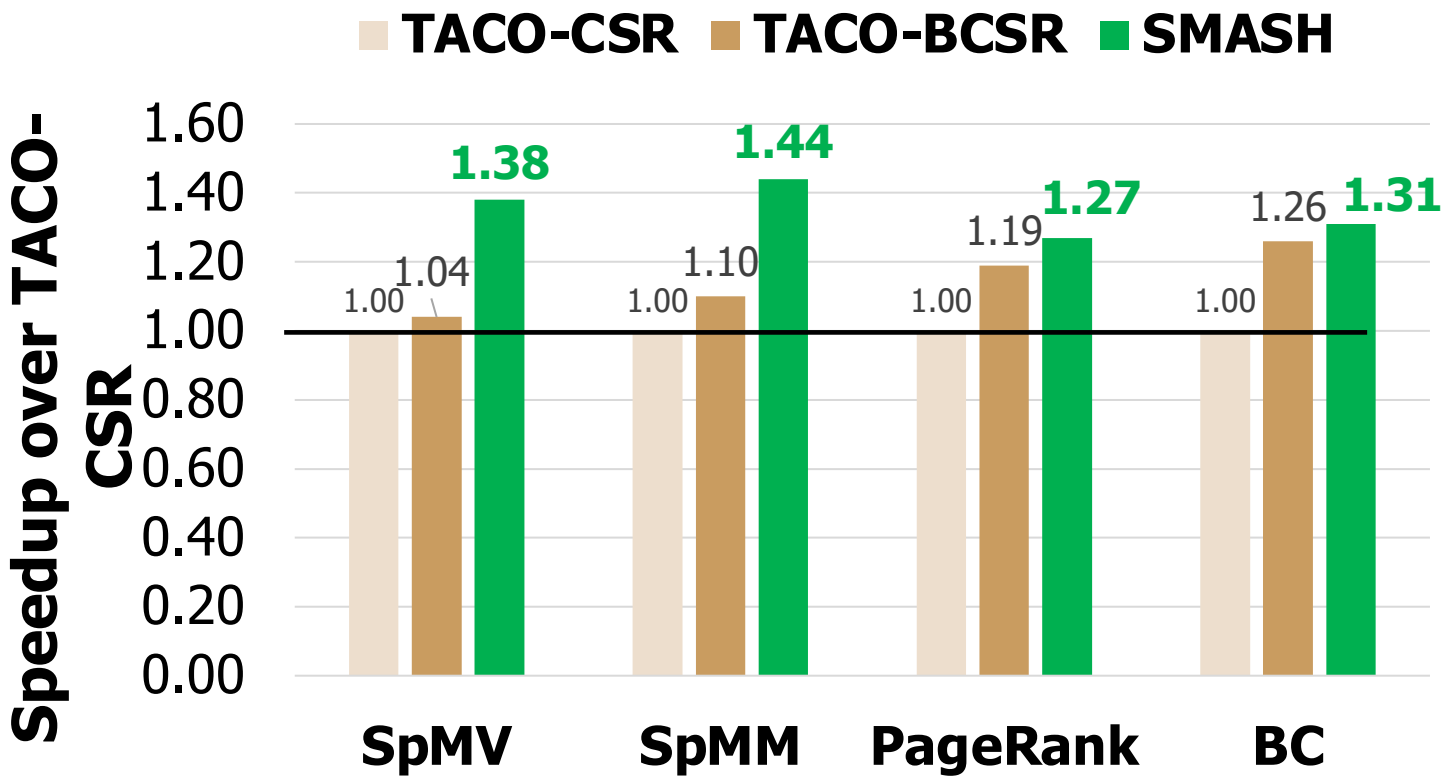
Idea: Apply the same encoding recursively to compress more effectively

Hierarchy of Bitmaps



Storage Efficient	Fast Indexing	Hardware Friendly

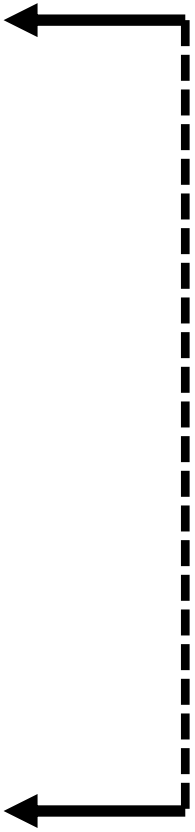
Performance Improvement Using SMASH



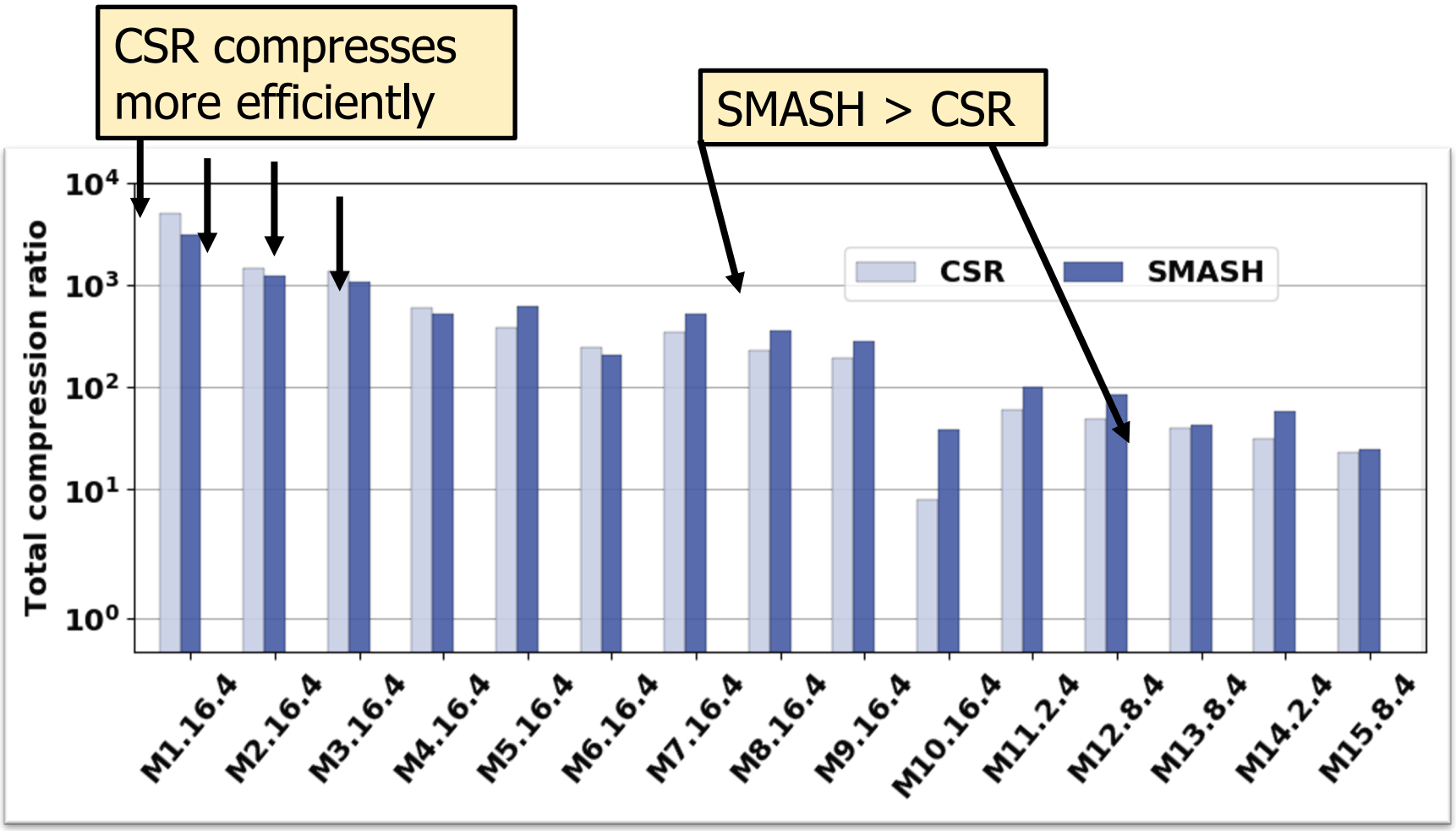
SMASH provides significant performance improvement over state-of-the-art formats
(even software-only SMASH outperforms TACO-CSR)

Evaluated Sparse Matrices

	Name	# Rows	Non-Zero Elements	Sparsity (%)
M1:	descriptor_xingo6u	20,738	73,916	0.01
M2:	g7jac060sc	17,730	183,325	0.06
M3:	Trefethen_20000	20,000	554,466	0.14
M4:	IG5-16	18,846	588,326	0.17
M5:	TSOPF_RS_b162_c3	15,374	610,299	0.26
M6:	ns3Da	20,414	1,679,599	0.40
M7:	tsyl201	20,685	2,454,957	0.57
M8:	pkustk07	16,860	2,418,804	0.85
M9:	ramage02	16,830	2,866,352	1.01
M10:	pattern1	19,242	9,323,432	2.52
M11:	gupta3	16,783	9,323,427	3.31
M12:	nd3k	9,000	3,279,690	4.05
M13:	human_gene1	22,283	24,669,643	4.97
M14:	exdata_1	6,001	2,269,500	6.30
M15:	human_gene2	14,340	18,068,388	8.79



Storage Efficiency



SMASH for Efficient Sparse Matrix Operations

- Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez-Luna, and Onur Mutlu, **"SMASH: Co-designing Software Compression and Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations"**
  *Proceedings of the 52nd International Symposium on Microarchitecture (**MICRO**)*, Columbus, OH, USA, October 2019.
  [Slides (pptx) (pdf)]
  [Lightning Talk Slides (pptx) (pdf)]
  [Poster (pptx) (pdf)]
  [Lightning Talk Video (90 seconds)]
  [Full Talk Lecture (30 minutes)]

SMASH: Co-designing Software Compression and Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations

Konstantinos Kanellopoulos[1] Nandita Vijaykumar[2,1] Christina Giannoula[1,3] Roknoddin Azizi[1]
Skanda Koppula[1] Nika Mansouri Ghiasi[1] Taha Shahroodi[1] Juan Gomez Luna[1] Onur Mutlu[1,2]
[1]ETH Zürich [2]Carnegie Mellon University [3]National Technical University of Athens

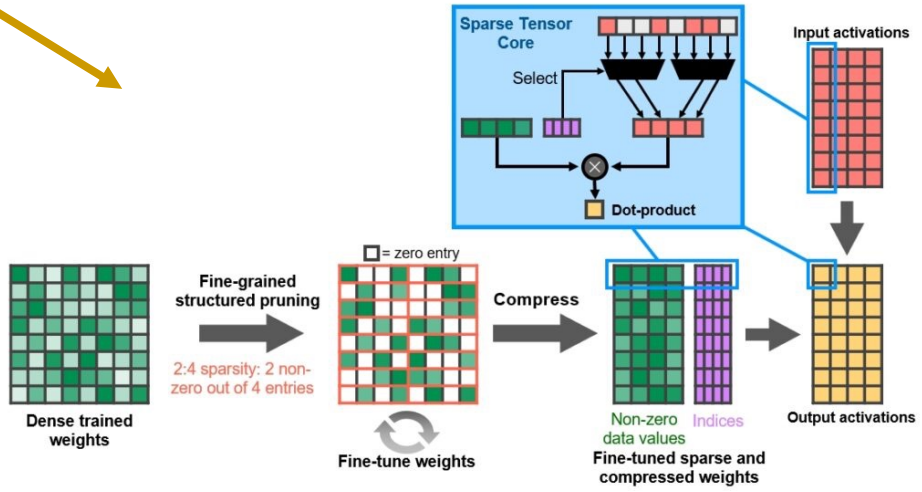
Using Tensor Core Units

Recall: NVIDIA A100 Core

19.5 TFLOPS Single Precision

9.7 TFLOPS Double Precision

312 TFLOPS for Deep Learning (Tensor cores)



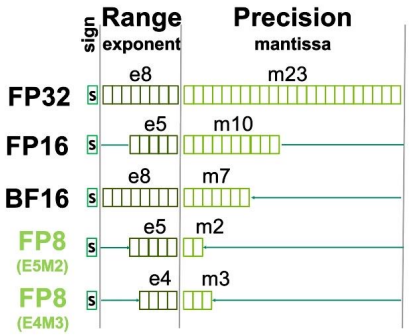
https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

Recall: NVIDIA H100 Core

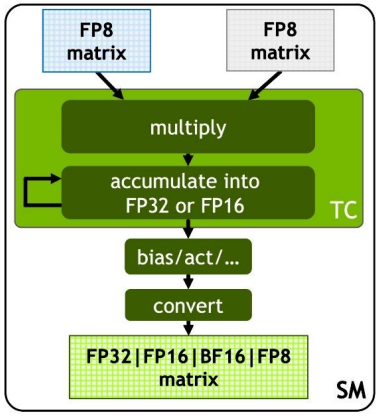
48 TFLOPS Single Precision*

24 TFLOPS Double Precision*

800 TFLOPS (FP16, Tensor Cores)*



Allocate 1 bit to either range or precision

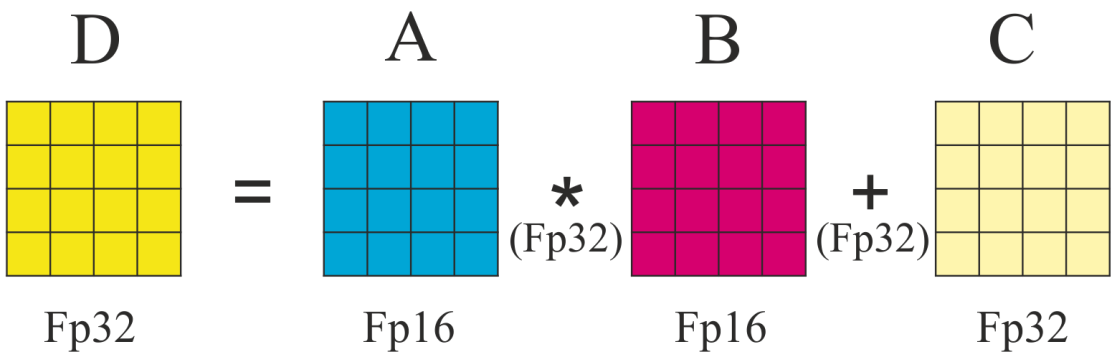


Support for multiple accumulator and output types

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

* Preliminary performance estimates

Characteristics of Tensor Core Units (TCUs)



$$D = A \times B + C$$

or

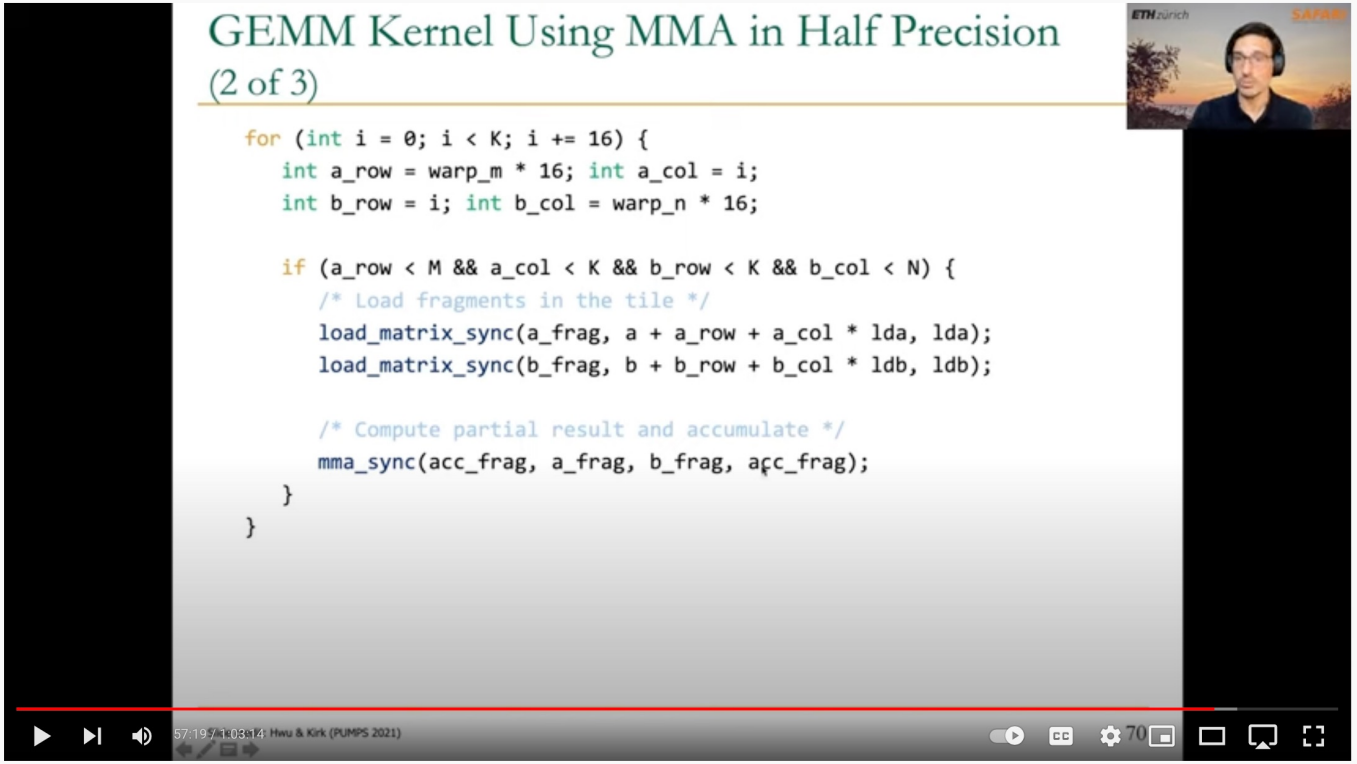
$$C = A \times B + C$$

Definitions

- TCUs perform matrix multiplication between "small" matrices
- "Small" is, e.g., 16x16
- Mixed precision: Result/Accumulator can be in FP32

Slide credit: Orestis Zachariadis

GEMM using Tensor Core Units (TCUs)



HetSys Course: Lecture 8: Parallel Patterns: Convolution (Spring 2022)

469 views • Premiered May 3, 2022

20 DISLIKE SHARE CLIP SAVE ...

Onur Mutlu Lectures
24.6K subscribers

SUBSCRIBED

Project & Seminar, ETH Zürich, Spring 2022
Hands-on Acceleration on Heterogeneous Computing Systems (
https://safari.ethz.ch/projects_and_s...)

Lecture 8: Parallel Patterns: Convolution
Lecturer: Dr. Juan Gómez Luna
Lecturer: Professor Onur Mutlu (https://people.inf.ethz.ch/omutlu/)
Date: May 3, 2022

https://youtu.be/-Tt9OFAXG78

Sparse Matrix Representation: COO

y\x	0	1	2	3
0	0	10	20	0
1	15	0	0	0
2	0	0	0	0
3	40	50	0	0

Row indices	0	0	1	3	3
Column indices	1	2	0	0	1
Values	10	20	15	40	50

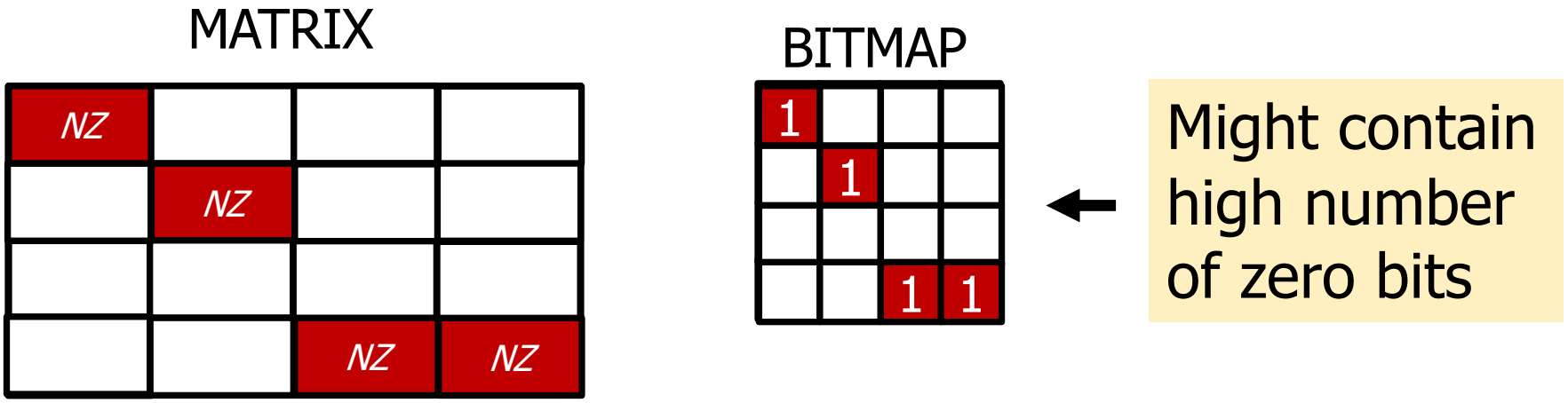
In this representation for every non-zero element we store its coordinates and its value

Slide credit: Orestis Zachariadis

SMASH: Software Compression Scheme (II)

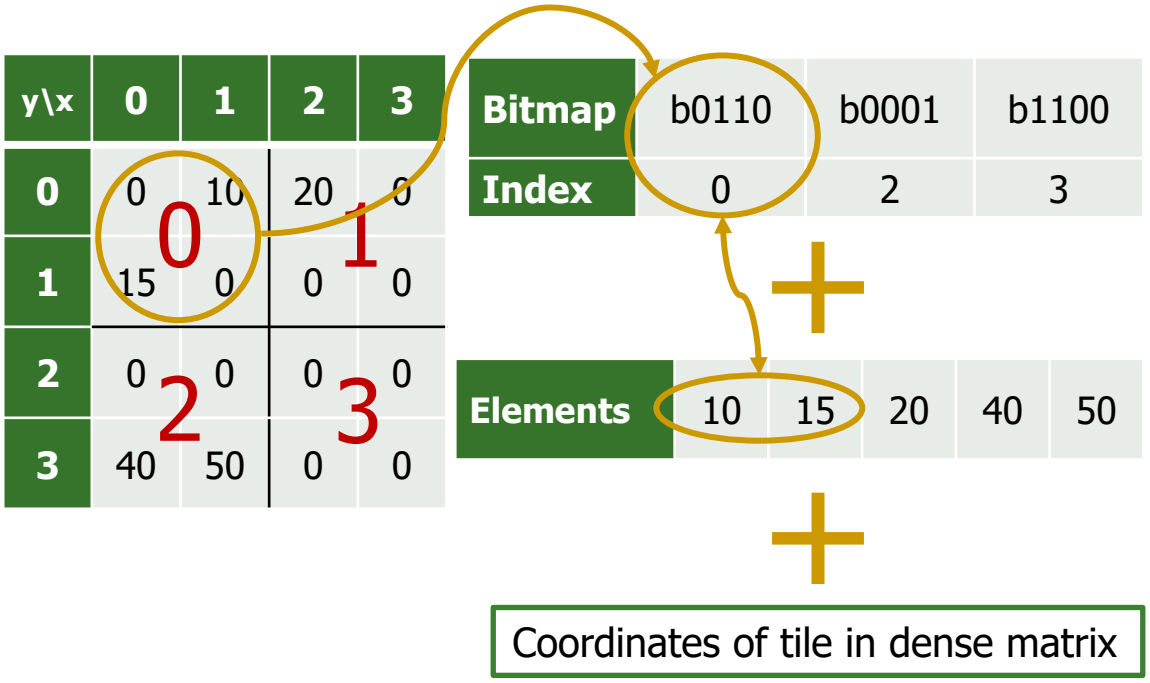
BITMAP:

Encodes if a block of the matrix contains any non-zero element



Idea: Apply the same encoding recursively to compress more effectively

Sparse Matrix Representation: Bitmap Format



Tiles

- A dense matrix is split into tiles

Bitmap

- A binary number for each tile. Each bit of is set to "1" iff there is an element in the corresponding position

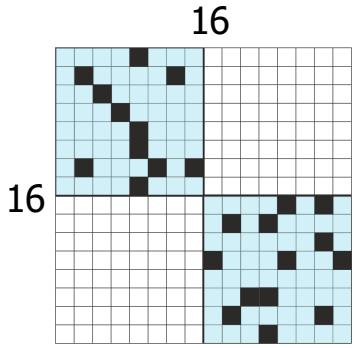
Index

- An offset in the element array. It points to the starting position of the elements of each tile

J. Zhang and L. Gruenwald, "Regularizing Irregularity : Bitmap-based and Portable Sparse Matrix Multiplication for Graph Data on GPUs," GRADES-NDA 2018

Slide credit: Orestis Zachariadis

How to Load Sparse Bitmaps into TCUs?



Nonzeros colored black

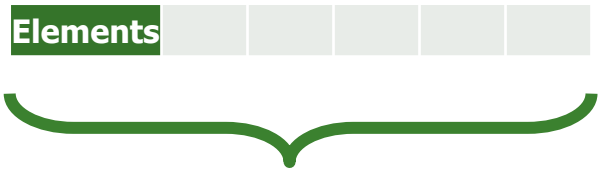
TCU layout

- We use 8x8 tiles
- Each warp multiplies 2 8x8 tiles
- Tensor cores are efficient even if not fully utilized[1]

[1] Dakkak et al., "Accelerating Reduction and Scan Using Tensor Core Units," ICS 2019

Slide credit: Orestis Zachariadis

Allocating Memory for the Output

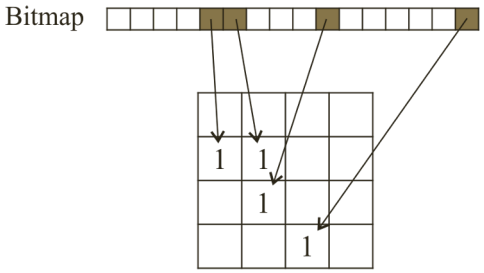


How many elements?

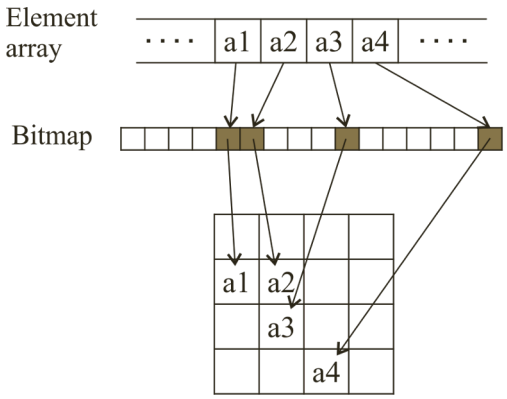
Memory pre-allocation

- Before executing the matrix multiplication we need to allocate memory
- We know how much memory to allocate only after the multiplication
- Various solutions: Upper bound, probabilistic, precise, progressive

Counting kernel



Multiplication kernel



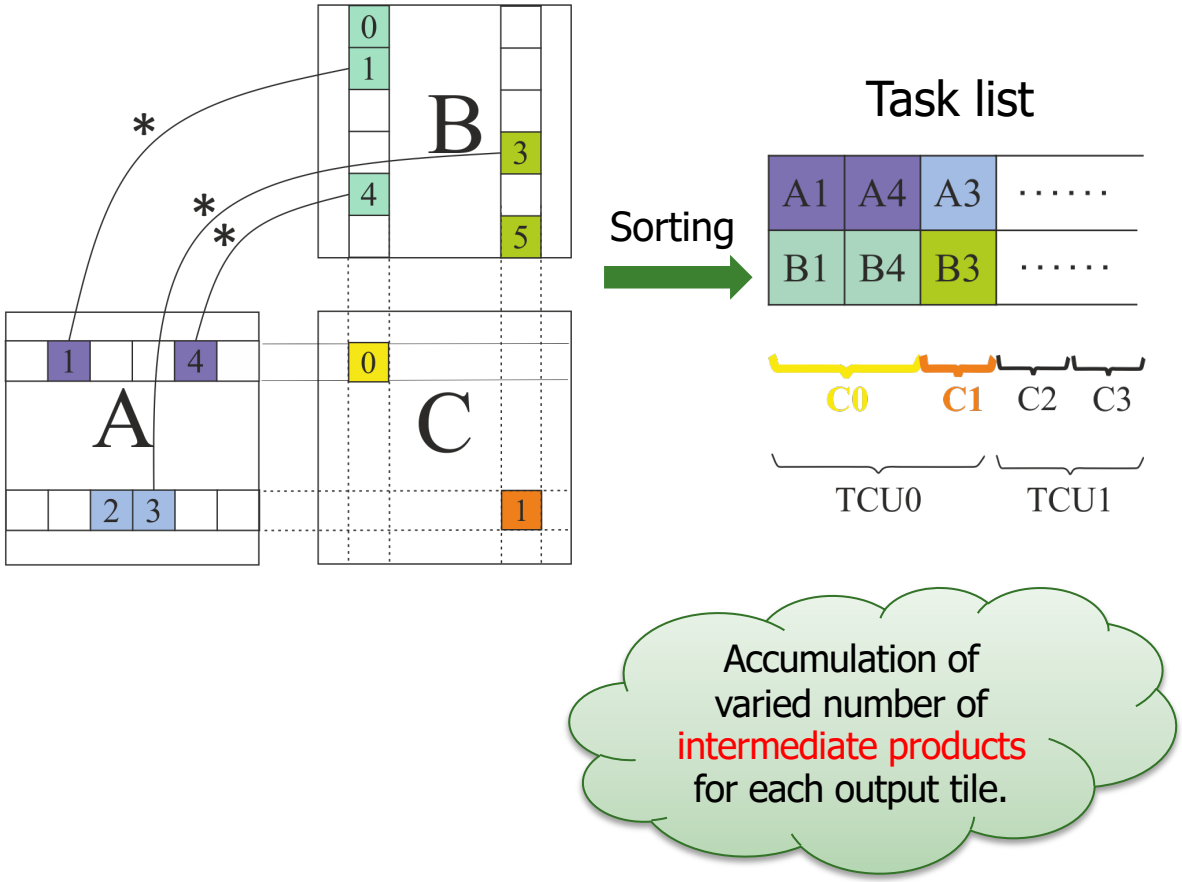
Precise method

- We estimate the count of elements in the output with a partial multiplication
- We do not load any actual values, instead we use only the bitmaps
- Similar code base, TCUs, etc.

Slide credit: Orestis Zachariadis

Finding Tiles to Multiply

Finding which tiles to multiply



Accumulation of varied number of intermediate products for each output tile.

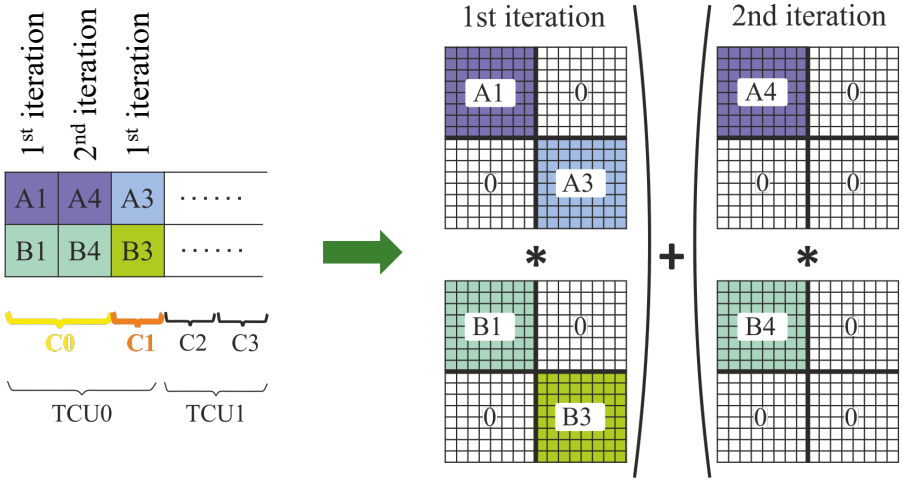
We perform the matrix multiplication as several tile multiplications of the form C = A*B + C

Finding matrix multiplications

1. We find which tiles of A need to be multiplied with which tiles of B for each tile of C
2. We group intermediate products using segmented sort algorithms
3. We create a task list of tiles to multiply

Slide credit: Orestis Zachariadis

Multiplying with TCUs (I)

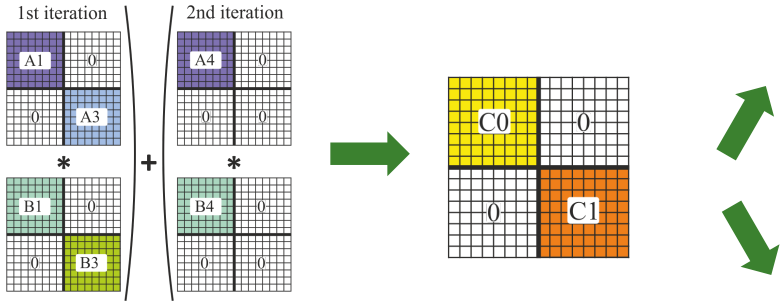


Multiplying tiles

- The task list feeds the TCUs
- We calculate 2 tiles of the output with each TCU
- We iterate until all intermediate products of both the inner products are accumulated

Slide credit: Orestis Zachariadis

Multiplying with TCUs (II)



Warp-wide ballot

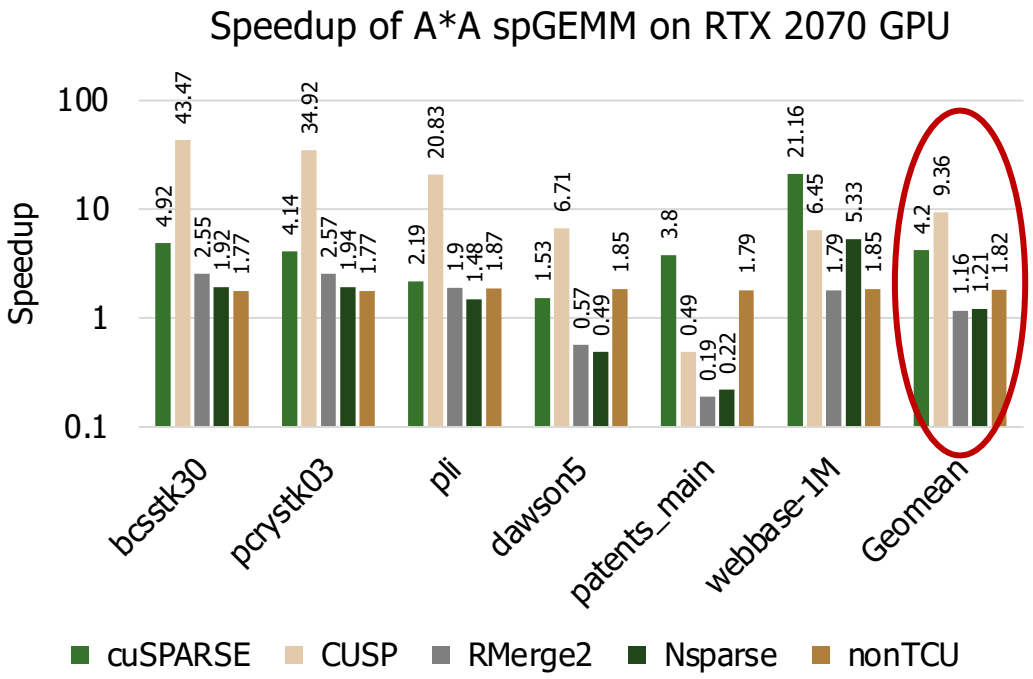
Image: Getty images

Bitmap C0	b0110	b0001	b1100
Bitmap C1	b0010	b1001	

Elements C0	10	15	20	40	50
Elements C1	5	13	7		

Slide credit: Orestis Zachariadis

Speedup of the Proposed Method (tSparse)



Observations

- On average 4.2x, 9.36x, 1.16x, 1.21x faster than cuSPARSE, CUSP, RMerge2, Nsparse, respectively
- TCUs are effective (1.82x faster than nonTCU)
- tSparse works better with denser tiles (density > 6)

Slide credit: Orestis Zachariadis

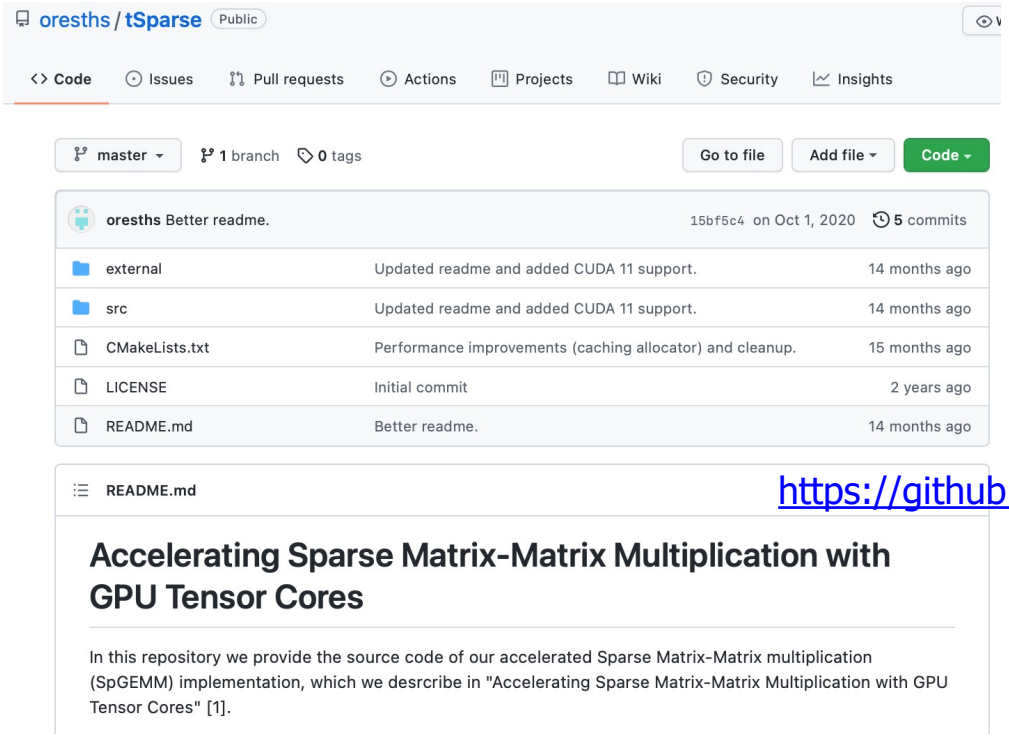
tSparse for SpGEMM

Accelerating sparse matrix–matrix multiplication with GPU Tensor Cores ☆

Orestis Zachariadis [a,*], Nitin Satpute [a], Juan Gómez-Luna [b], Joaquín Olivares [a]

[a] Department of Electronic and Computer Engineering, Universidad de Cordoba, Córdoba, Spain
[b] Department of Computer Science, ETH Zurich, Zurich, Switzerland

Zachariadis et al., "Accelerating Sparse Matrix–Matrix Multiplication with GPU Tensor Cores," Computers and Electrical Engineering, 2020 [arXiv version: http://arxiv.org/abs/2009.14600]



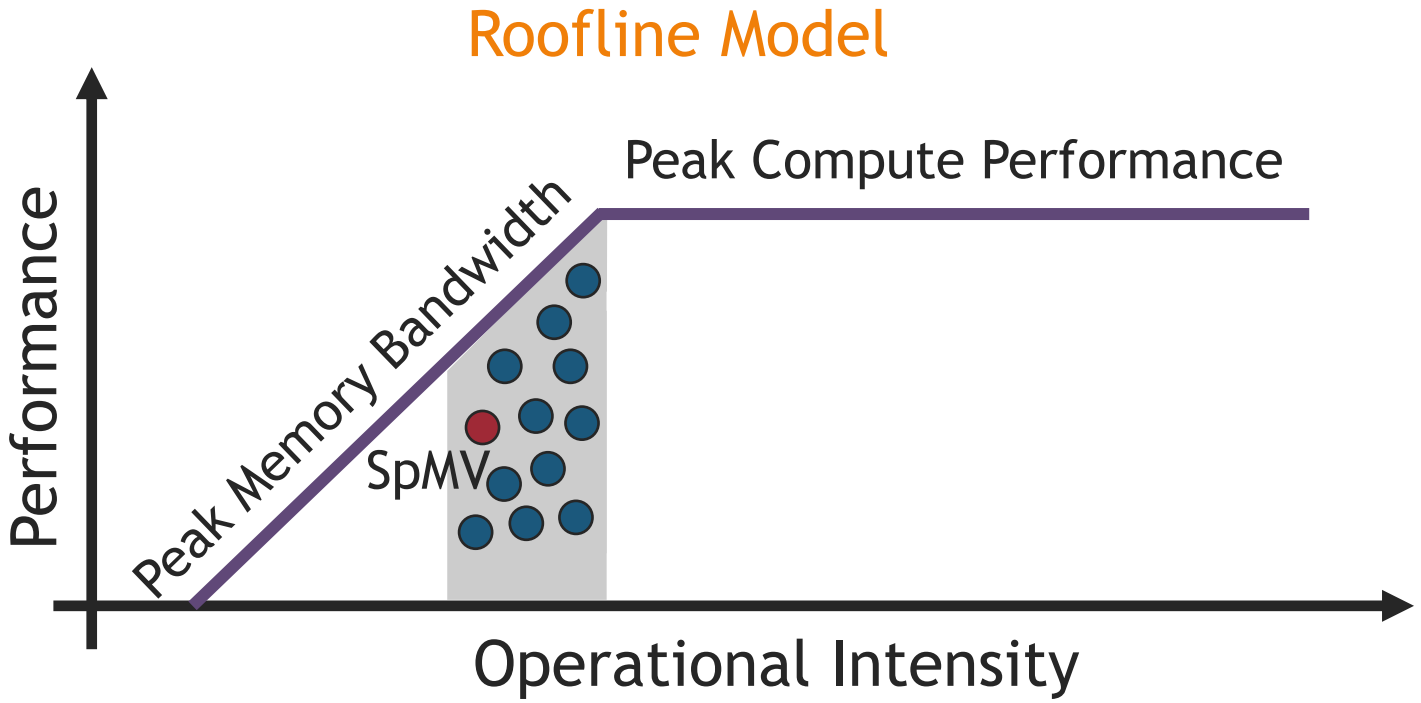
https://github.com/oresths/tSparse

SpMV is Memory Bound

Sparse Matrix Vector Multiplication

Sparse Matrix Vector Multiplication (SpMV):

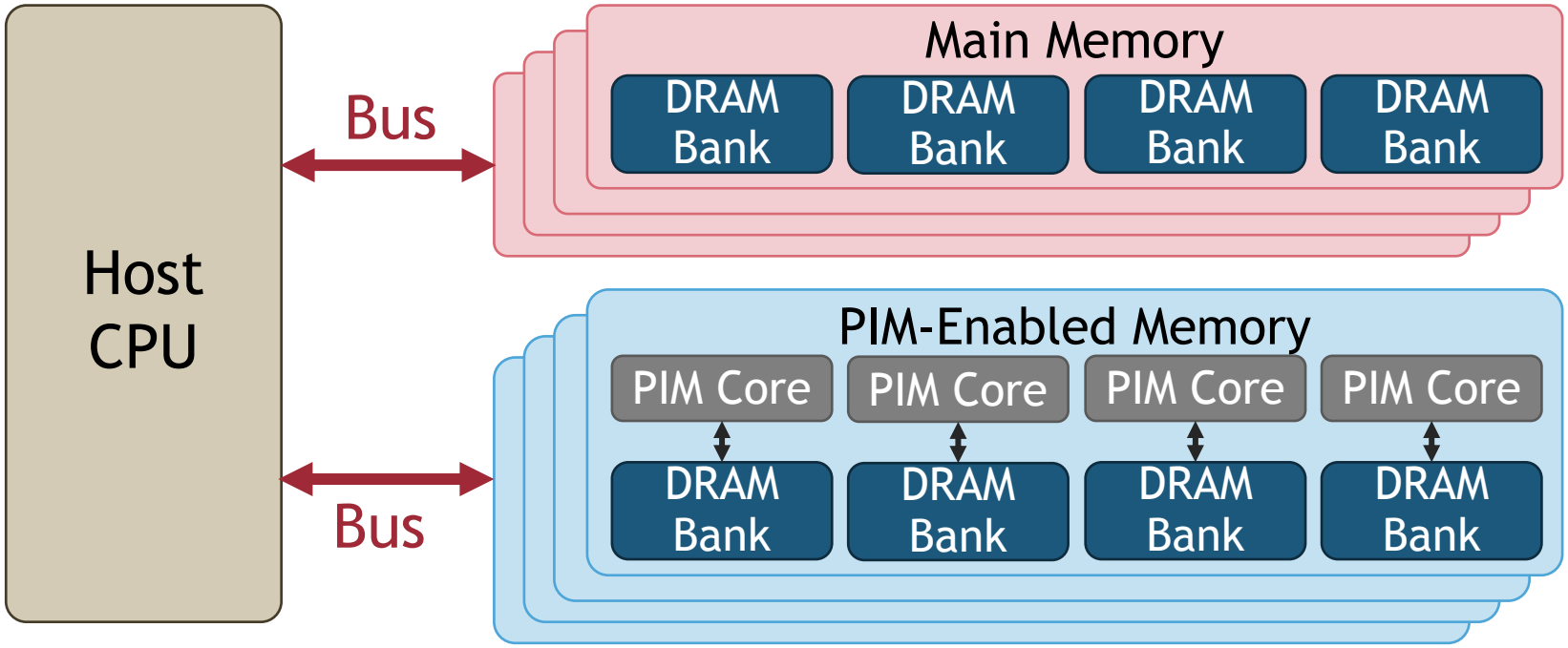
- Widely-used kernel in graph processing, machine learning, scientific computing …
- A highly memory-bound kernel



Real Processing-In-Memory Systems

Real Near-Bank Processing-In-Memory (PIM) Systems:

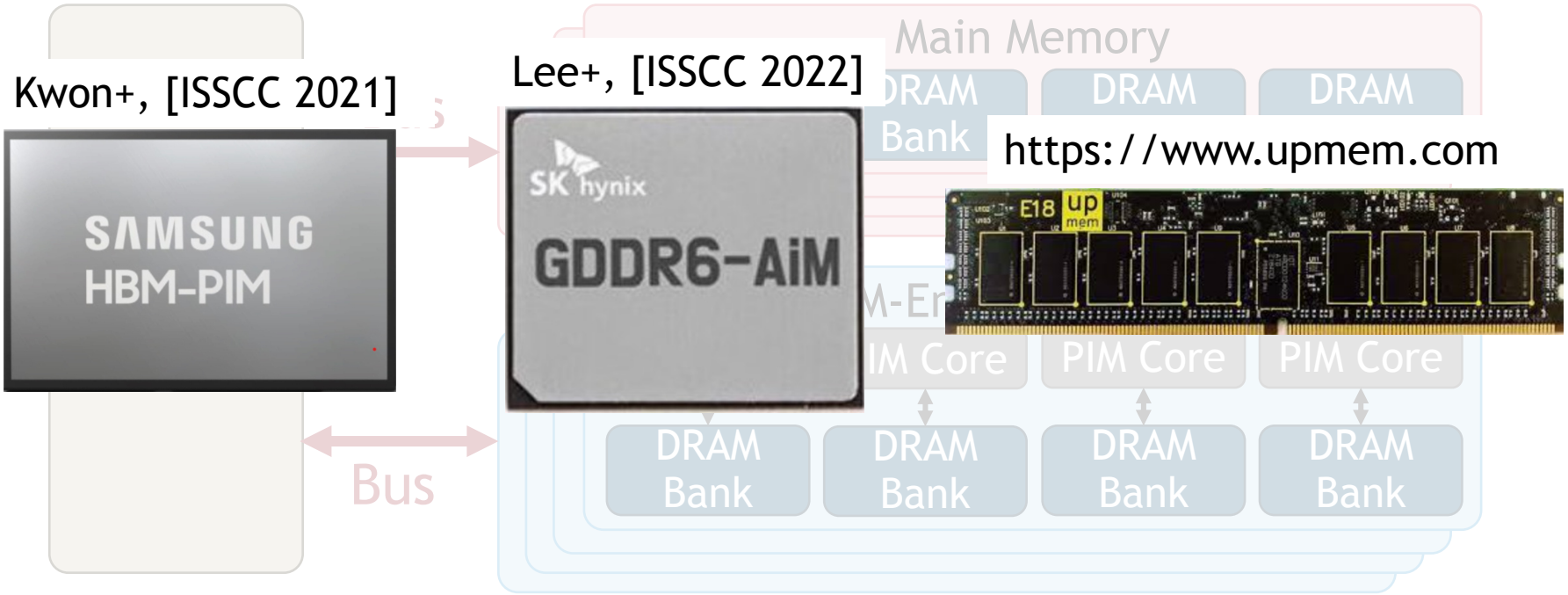
- High levels of parallelism
- Low memory access latency
- Large aggregate memory bandwidth



Real Processing-In-Memory Systems

Real Near-Bank Processing-In-Memory (PIM) Systems:

- High levels of parallelism
- Low memory access latency
- Large aggregate memory bandwidth



SparseP: SpMV Library for Real PIMs

Our Contributions:

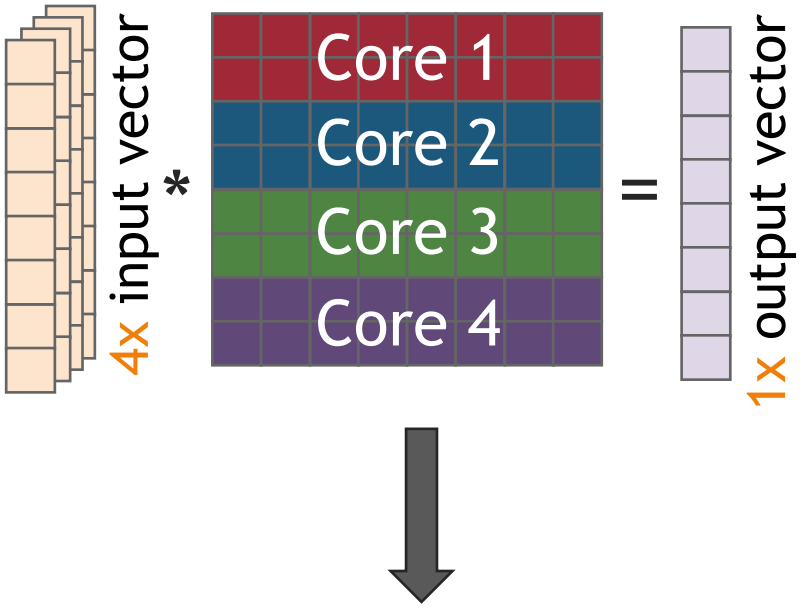
1. Design efficient SpMV kernels for current and future PIM systems
   - 25 SpMV kernels
     - 4 compressed matrix formats (CSR, COO, BCSR, BCOO)
     - 6 data types
     - 4 data partitioning techniques
     - Various load balancing schemes among PIM cores/threads
     - 3 synchronization approaches

2. Provide a comprehensive analysis of SpMV on the first commercially-available real PIM system **upmem**
   - 26 sparse matrices
   - Comparisons to state-of-the-art CPU and GPU systems
   - Recommendations for software, system and hardware designers

Data Partitioning Techniques

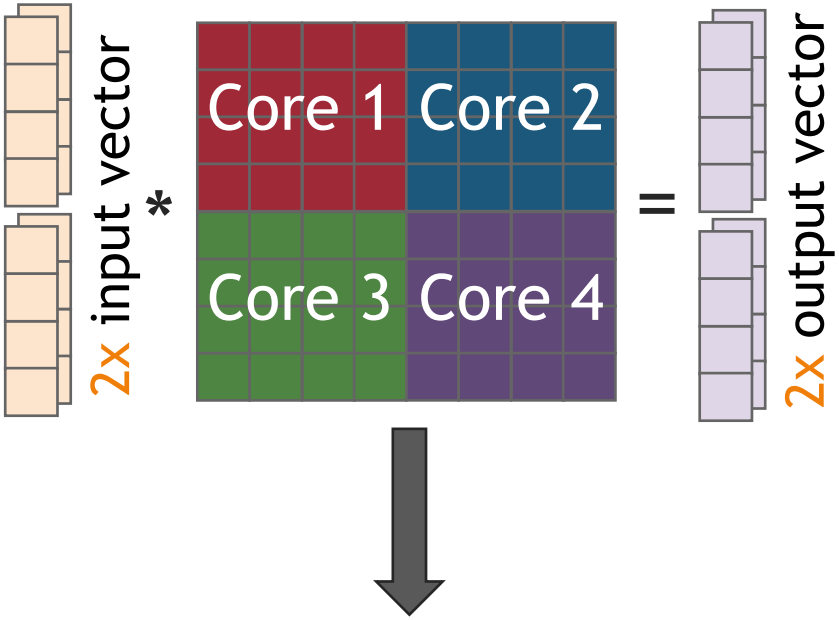
SparseP supports two types of data partitioning techniques:

1D Partitioning



perform the complete SpMV computation only on PIM cores

2D Partitioning



trade-off computation vs data transfer costs

SparseP Software Package

25 SpMV kernels for PIM Systems →
https://github.com/CMU-SAFARI/SparseP

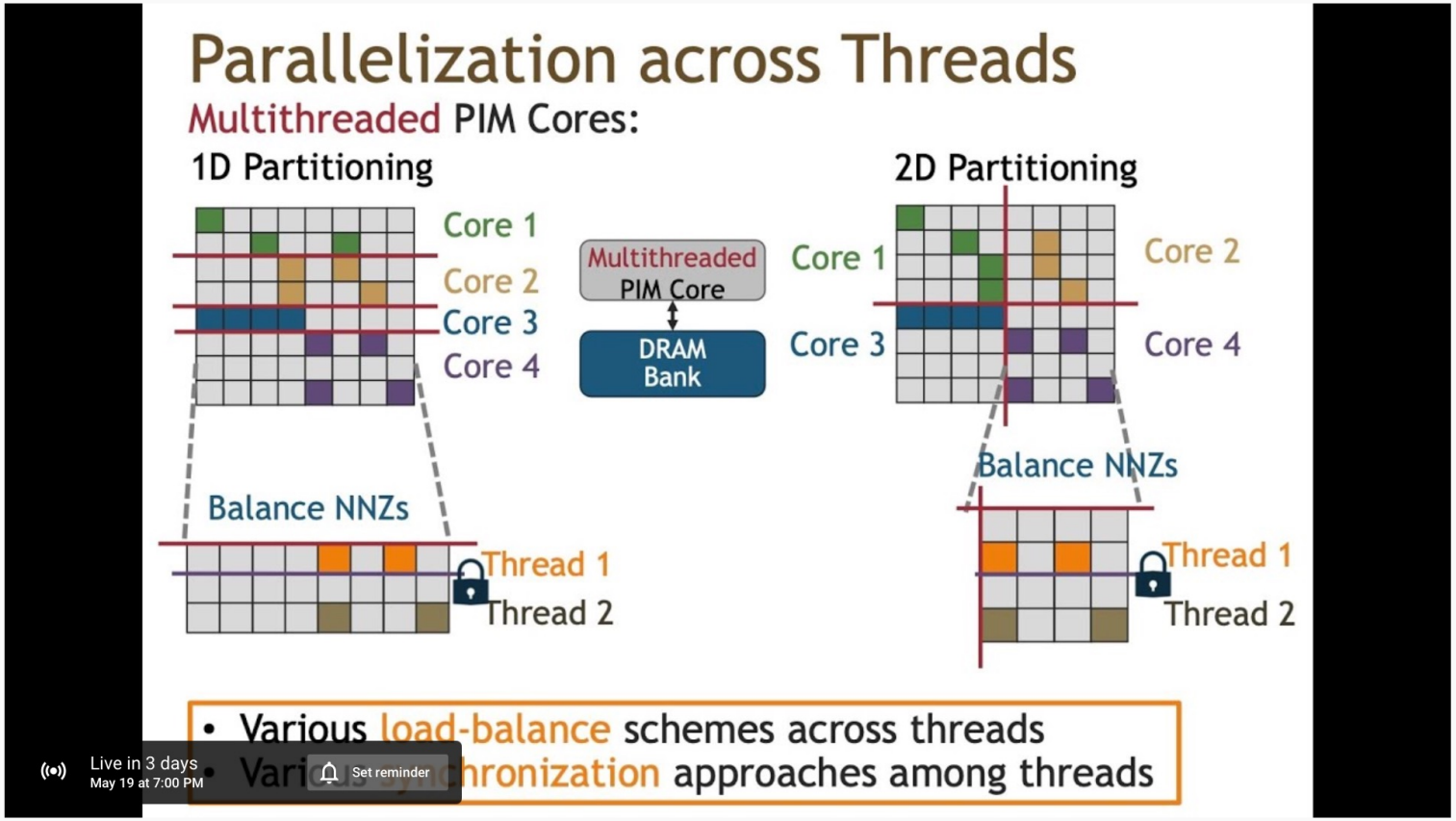
Partitioning	Matrix Format	Load-Balancing
9x 1D Kernels	CSR	rows, nnzs *
	COO ▲	rows, nnzs *, nnzs
	BCSR	blocks ^, nnzs ^
	BCOO ▲	blocks, nnzs
4x 2D Equally-Sized Tiles	CSR	--
	COO ▲	--
	BCSR	--
	BCOO ▲	--
6x 2D Equally-Wide Tiles	CSR	nnzs *
	COO ▲	nnzs
	BCSR	blocks ^, nnzs ^
	BCOO ▲	blocks, nnzs
6x 2D Variable-Sized Tiles	CSR	nnzs *
	COO ▲	nnzs
	BCSR	blocks ^, nnzs ^
	BCOO ▲	blocks, nnz

Load-balance
across PIM cores/threads:
* row-granularity (CSR)
^ block-row-granularity (BCSR)

Synchronization
among threads of a PIM core:
▲ lb-cg, lb-fb, lf (COO, BCOO)

Data Types:
- 8-bit integer
- 16-bit integer
- 32-bit integer
- 64-bit integer
- 32-bit float
- 64-bit float

Lecture about SpMV on Processing-in-Memory



Processing-in-Memory Course: Lecture 11: SpMV on a Real PIM Architecture - Spring 2022

1 waiting • Scheduled for May 19, 2022

1 DISLIKE SHARE SAVE ...

Onur Mutlu Lectures

24.8K subscribers

SUBSCRIBED

Projects & Seminars, ETH Zürich, Spring 2022

Exploring the Processing-in-Memory Paradigm for Future Computing Systems (

https://youtu.be/5kaOsJKlGrE

SparseP: SpMV on Processing-in-Memory

SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Systems

CHRISTINA GIANNOULA, ETH Zürich, Switzerland and National Technical University of Athens, Greece
IVAN FERNANDEZ, ETH Zürich, Switzerland and University of Malaga, Spain
JUAN GÓMEZ-LUNA, ETH Zürich, Switzerland
NECTARIOS KOZIRIS, National Technical University of Athens, Greece
GEORGIOS GOUMAS, National Technical University of Athens, Greece
ONUR MUTLU, ETH Zürich, Switzerland

Several manufacturers have already started to commercialize near-bank Processing-In-Memory (PIM) architectures, after decades of research efforts. Near-bank PIM architectures place simple cores close to DRAM banks. Recent research demonstrates that they can yield significant performance and energy improvements in parallel applications by alleviating data access costs. Real PIM systems can provide high levels of parallelism, large aggregate memory bandwidth and low memory access latency, thereby being a good fit to accelerate the Sparse Matrix Vector Multiplication (SpMV) kernel. SpMV has been characterized as one of the most significant and thoroughly studied scientific computation kernels. It is primarily a memory-bound kernel with intensive memory accesses due its algorithmic nature, the compressed matrix format used, and the sparsity patterns of the input matrices given.

This paper provides the first comprehensive analysis of SpMV on a real-world PIM architecture, and presents *SparseP*, the first SpMV library for real PIM architectures. We make three key contributions. First, we implement a wide variety of software strategies on SpMV for a multithreaded PIM core, including (1) various compressed matrix formats, (2) load balancing schemes across parallel threads and (3) synchronization approaches, and characterize the computational limits of a single multithreaded PIM core. Second, we design various load balancing schemes across multiple PIM cores, and two types of data partitioning techniques to execute SpMV on thousands of PIM cores: (1) 1D-partitioned kernels to perform the complete SpMV computation only using PIM cores, and (2) 2D-partitioned kernels to strive a balance between computation and data transfer costs to PIM-enabled memory. Third, we compare SpMV execution on a real-world PIM system with 2528 PIM cores to an Intel Xeon CPU and an NVIDIA Tesla V100 GPU to study the performance and energy efficiency of various devices, i.e., both memory-centric PIM systems and conventional processor-centric CPU/GPU systems, for the SpMV kernel. *SparseP* software package provides 25 SpMV kernels for real PIM systems supporting the four most widely used compressed matrix formats, i.e., CSR, COO, BCSR and BCOO, and a wide range of data types. *SparseP* is publicly and freely available at https://github.com/CMU-SAFARI/SparseP. Our extensive evaluation using 26 matrices with various sparsity patterns provides new insights and recommendations for software designers and hardware architects to efficiently accelerate the SpMV kernel on real PIM systems.

https://arxiv.org/pdf/2201.05072.pdf

SparseP Repository

main · 1 branch · 0 tags · Go to file · Add file · Code

cgiannoula Initial commit — 5c5875e on Feb 7 — 9 commits

images	Initial commit	3 months ago
inputs	Initial commit	3 months ago
scripts	Initial commit	3 months ago
spmv	Initial commit	3 months ago
LICENSE	Initial commit	3 months ago
README.md	Initial commit	3 months ago

README.md

SparseP Software Package v1.0

About

SparseP is the first open-source Sparse Matrix Vector Multiplication (SpMV) software package for real-world Processing-In-Memory (PIM) architectures. SparseP is developed to evaluate and characterize the first publicly-available real-world PIM architecture, the UPMEM PIM architecture. Described by C. Giannoula et al. [https://arxiv.org/abs/2201....

Readme
MIT license
23 stars
4 watching
5 forks

Releases

No releases published
Create a new release

Packages

No packages published
Publish your first package

Contributors 2

cgiannoula Christina Giannoula

https://github.com/CMU-SAFARI/SparseP

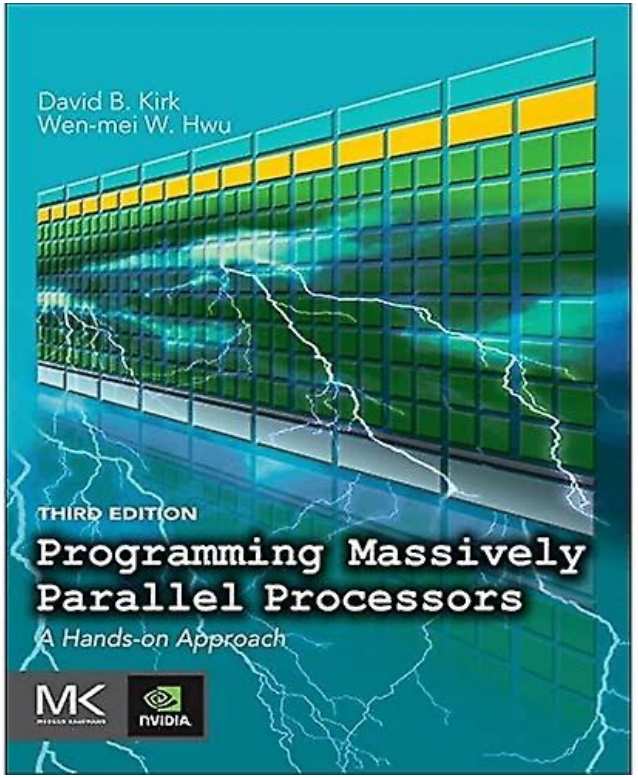
Recommended Readings (I)

- Hwu and Kirk, "Programming Massively Parallel Processors," Third Edition, 2017
  - Chapter 10 - Parallel patterns:

  sparse matrix computation:

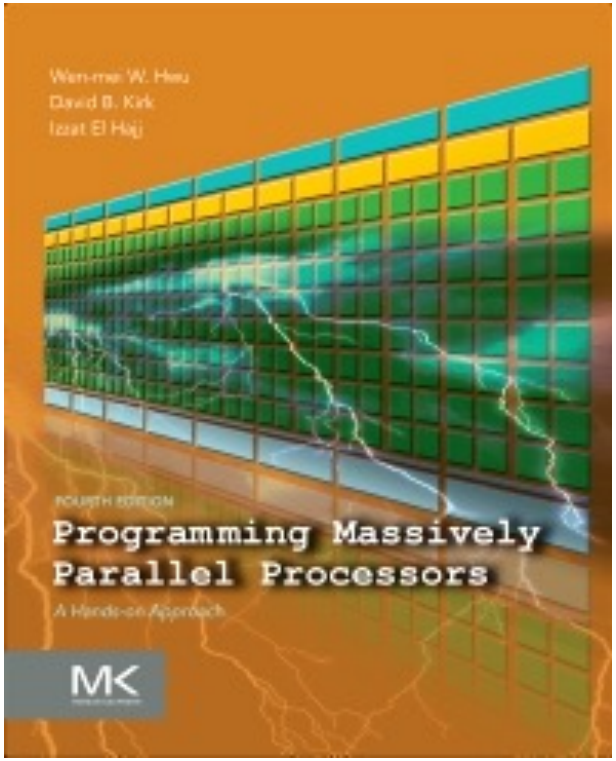
  An introduction to data compression

  and regularization



Recommended Readings (II)

- Hwu and Kirk and El Hajj, "Programming Massively Parallel Processors," Fourth Edition, 2022
  - Chapter 14 - Sparse matrix computation



P&S Heterogeneous Systems

Parallel Patterns: Sparse Matrices

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich

Fall 2022

12 December 2022